A Reliable Network Boot Service for PCs

by

BHARTENDU SINHA

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

July, 1996

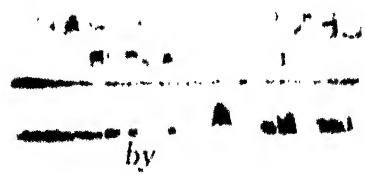
A Reliable Network Boot Service for PCs

A Thesis Submitted

in Partial Fulfillment of the Requirements

for the Degree of

Master of Technology



by

Bhartendu Sinha

to the

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

July 1996

26/7/96

CERTIFICATE

This is to certify that the work contained in the thesis entitled A Reliable Network Boot Service for PCs by Bhartendu Sinha has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Dr. Rajat Moona,
Associate Professor,
Department of Computer Science & Engineering,
Indian Institute of Technology, Kanpur.

Abstract

Remote booting of a PC, using its network interface instead of local disk drives, offers significant advantages to PC users, to system administrators, and also to system designers. The basis of remote boot is the use of an extension ROM, called a *boot* ROM, the availability of boot servers on the network, and the existence of a boot image.

In this thesis, existing implementations of PC remote boot have been studied, the remote boot procedure has been largely redesigned, and the new design has been fully implemented. The goal has been to make a reliable and robust remote boot product. In particular, the emphasis has been on overcoming existing bugs, providing boot management flexibility, using a simpler and more standard set of protocols, optimizing client code, data & stack size, transparent handling of any PC disk configuration, providing multiple ethernet card support, ensuring portability across servers, handling security issues and giving improved diagnostics. Equal emphasis has been put on keeping a high standard of code quality, and on providing complete documentation of the product from all perspectives.

Acknowledgements

I am grateful to my thesis supervisor, Dr. Rajat Moona, who allowed me to choose a problem which I felt would be challenging and add substantially to my knowledge. Thanks to him, I now have an understanding of the implementation of networking protocols from the ethernet layer to the application layer, as well as a much improved knowledge of DOS and BIOS internals. When I would be unable to diagnose a problem, Dr. Moona always had a solution, an insight, or some guidelines to offer. More importantly, he set for me an example of technical excellence which I hope to follow.

I would like to thank the staff of the CSE lab, who were always patient with my demands on their time, and had enough trust in me to allow me to meddle with the system configuration & hardware - both patience and trust being rare qualities in system administrators.

I must thank my friends who tolerated my long absences from their midst, and provided me with much needed entertainment when I would be with them.

And also, I have to thank God for providing inspiration when all else seemed to have failed.

Contents

Chapter 1

Introduction

1.1 Motivation

The concept of stand-alone computing is fast becoming redundant. A networking interface is now accepted as an integral part of modern computing systems. This is because networking allows distributed computing, using which the cost of a computing facility can be significantly reduced by the sharing of resources. Such sharing has only a marginal effect on the availability of resources to any one user. Resource sharing has been successful for a variety of software services and hardware I/O peripherals.

Personal Computers (PCs) are the most widely used computational platform. In a distributed environment, PCs can share common file systems and no longer need to store files locally. However, PCs have an intricate booting procedure design, which is dependent on the existence of a local floppy disk or hard disk. If this dependency on local disks for booting can be bypassed by booting over the network, truly networked PCs can be be created.

Simply booting over the network is not sufficient. Equally important is to make the remote boot of PCs fully transparent and reliable. Primarily, this means that the response of the PCs during and after the remote boot is identical to the response of the PCs during and after a local boot. It also means several other things, as are discussed in this thesis.

Thus, the motivation of this thesis is to develop a remote boot product for PCs which can be easily and reliably implemented on any set of PCs in a networked environment, without affecting the behavior of the system in any adverse manner.

1.2 The Concept of Remote Boot

The process of bringing a computer system up into an operational state, after powering it up, is called booting. The goal of booting is to initialize the hardware components and then get a copy of the operating system into memory. Remote booting of a computer is the booting of a computer using its network interface instead of its local disks. The core technology behind this concept is to use an extension ROM, called a *boot* ROM, which can be added to any computer's network interface card. A code in this ROM relies on the availability of specific servers on the network, and the existence of a boot file, i.e. a binary image of a boot floppy. While booting, BIOS transfers control to the program in the boot ROM, which communicates with the servers to access the boot file over the network. Together, they enable any PC to boot over the network, without requiring any changes other than the addition of a boot ROM in the network interface card, the existence of certain servers on the network, the creation of a boot file, and the management of servers to allow a PC to access its boot file.

The routines in the boot ROM are quite complicated. They must support networking protocols from the medium access control (MAC) layer to the application layer, they must not use any operating system calls - none being available during the booting process, they must fit into a standard size boot ROM, they cannot be tested using debuggers, etc. As such, the development of a boot ROM is a very different task from the development of general application or system software.

Given the complexity and criticality of the boot ROM design, it becomes essential to follow high quality software engineering guidelines in its development. This greatly reduces the cost of future maintenance and upgradation of the code.

Remote booting offers significant advantages to computer users, to system administrators, and also to system designers. All workstations have built in support

for remote boot (also referred to as network bootstrap). Therefore, the advantages discussed below are relevant to PCs, which do not have any standard built-in support for remote boot.

1.2.1 Advantages to Users

- With todays high speed networks, network file access is significantly faster than floppy disk file access. Thus remote boot takes much less time than a floppy disk boot - although a bit more time than a hard disk boot.

- Users no longer need to worry about the virus problems associated with the multiple and unprotected usage of floppy disks for booting, thus saving them time and money.

- The users no longer need to worry about version incompatibility between their boot files and the files stored on the network drives. This job gets centralized with the system administrator.

1.2.2 Advantages to System Administrators

- The system administrator's job of providing a controlled computing environment becomes centralized and simple. Only the boot files and some configuration files need to be changed to control the computing environment.

- With the users' problems of virus attacks and software version incompatibility centralized at one point, i.e. at the boot file, the system administrator will receive far fewer complaints and maintenance requests, thus saving time for more important work.

- As a PC can boot and be operational even if its disk drives have malfunctioned, such problems can be given a lower priority and handled in batches rather than immediately. It should be noted that most PC boot failures are generally due to bad drives or corrupted boot disks. Networks today are highly reliable, and thus remote boot usage can avoid most cases of boot failure.

- The user activity on PCs can be recorded, as when the PC is booting the user can be forced to authenticate with a server. This discourages misuse of the systems, and allows misuse to be traced.

1.2.3 Advantages to System Designers

- The major advantage of remote boot is that it gives feasibilty to a diskless PC configuration. With the present demands on the size, weight, power and cost of PCs, eliminating disks by using remote boot can give system designers a significant competitive advantage.

- Remote boot is a particularly attractive component in mobile PCs, which must be low weight, low power, small and cheap. If the PC has no local disks, it helps to meet the above requirements.

- Networks today provide highly a reliable I/O service, whereas disks (in particular floppy disks) are relatively error prone. As such, the reliability of remote boot is better than that of a local disk boot. Therefore, systems using remote boot will result in better availability of the system. This can be used to reduce the cost of built-in redundancy and maintenance overheads.

1.3 Design Goals and Implementation Features

The design goals and implementation features of this thesis are summarized here, to serve as a quick reference. They are explained in detail from chapter 3 onwards.

- **Server Design Goals:** The server design goals are to use or develop UNIX platform services (based on the TCP/IP protocol suite) to support remote boot. Simple boot management flexibility is to be provided for the system administrator. As far as possible, standard UNIX services are to be used, and non-standard services should be eliminated or merged. The non-standard data bases should be eliminated. The PCs' user login activity should be recorded in a secure and standard data base. Finally, all the server source code should be fully portable.

- **Server Implementation Features:** The server implementation uses the BOOTP protocol for access to boot initialization information. There is a UDP protocol to provide boot file access. This UDP protocol also provides an authentication service and it uses platform independent data types.

- **Client Design Goals:** The client design goals include adhering to a 16 kB boot ROM size limit. The PC should provide an unchanged user interface for booting (apart from the authentication). The client code must not use operating system calls. No changes should be required in the PC other than the addition of a boot ROM. The PC must maintain a reasonable boot performance in terms of time. The protocols used should be light weight protocols. The size of the code, data & stack in the ROM should be optimized. The client code should be able to handle any PC disk configuration transparently. The client code should also provide operating system independence. There should be multiple ethernet card support. Security issues should be properly addressed. The diagnostics should be properly designed. In addition, it should be possible to test the system without the creation of a boot ROM.

- **Client Implementation Features:** The client implementation uses BOOTP to obtain the boot information and proceeds on the basis of this information. UDP is used for boot file access. Techniques for optimization of code, data and stack size are implemented. Transparency to the PC disk configuration has been provided. The required DOS function calls have been rewritten. Care has been taken to allow for for operating system independence. Ethernet card probes have been implemented. An ne2000 card device driver has been written and integrated with the system. Steps have been taken to ensure password security. There is intelligent handling of errors and diagnostics. The implementation of a Terminate and Stay Resident (TSR) version has also been done.

- **Software Code Quality:** With a goal of producing high quality code, software quality guidelines were followed. These guidelines take into account file

and function headers, partitioning of header files, and macro, typedef, function & variable declarations. These guidelines also take into account design modularity, conventions for conditionals and control flow, and conventions for spacing, line wrap & comments.

- **Software Documentation Quality:** With a goal of producing high quality documentation, a `PC Remote Boot Installation Manual` was written dealing with all aspects of the installation of the remote boot system.

1.4 Overview of the Thesis

The rest of this thesis is organized as follows. In chapter 2 we explain the earlier framework, which is also the basis for the current work. Chapter 3 explains the design issues for the entire work. Chapters 4, 5 and 6 discuss the server, client and software quality implementation. Chapter 7 is about the remote boot file. Chapter 8 summarizes the conclusions and results. Finally, there is a comprehensive appendix providing guidelines for the implementation and installation of remote boot in a typical setup. Two distribution diskettes are supplied as part of the appendix.

Chapter 2

The Existing Framework

This thesis is a continuation of earlier work done on remote booting. As such, it is necessary to explain the earlier framework which was available. This framework relates to technical aspects of the remote boot of UNIX machines and personal computers, to the commercial background of the implementations that are available in industry, and to the work already done in this area at IIT Kanpur. In particular, it describes in some detail the thesis work done by T. J. Manujnath under Dr. R. Moona in 1994 [Man94], and some of the features of the remote boot system that was developed. This chapter is largely a condensation of the material presented in the above thesis by T. J. Manjunath. The chapter is lengthy as many of the key concepts involved in remote boot are explained here.

Reading of this chapter is essential to understand the starting point for the work done in this thesis, and also to understand the background and history of remote boot in general.

2.1 Remote Boot of UNIX Machines

The booting of UNIX workstations uses a simple and cleanly designed procedure, which makes it suitable for the implementation of remote boot. A special program, called boot, is loaded into the main memory and started running. On diskless UNIX workstations, this program resides in a ROM, but in general, it may be kept in any

storage device accessible to the workstation.

The `boot` program's task is to load the executable image of the UNIX kernel code. The UNIX kernel is a single program, whose binary image resides in a file, usually called `/unix` or `/vmunix`. The kernel file is loaded using a standard file transfer utility like TFTP [SC81], and the file is read from the file system of a server machine [Fin84].

After loading the UNIX kernel, the boot program hands over control to the kernel to complete the booting procedure.

Because of the simplicity of the booting procedure, diskless UNIX workstations fitted with a boot ROM to support remote boot are commonly available in the industry.

2.2 Remote Boot of Personal Computers

Unlike UNIX workstations, whose booting procedure makes implementing remote boot relatively simple, personal computers have a complicated booting procedure. This procedure involves multiple states, multiple operations and multiple files. There is also no built-in support in PCs for booting from any media other than their local disks. Due to these reasons, almost all networked PCs today still boot from their local disks.

To understand how to remote boot a PC, the earlier booting procedure must first be understood. A summary is presented here.

2.2.1 Startup Procedure by BIOS

When a PC is switched on, the program execution starts at a well defined location (`0xFFFF0`) in the ROM-BIOS. After the initial power on self test, hardware & interrupt vector table initialization, the BIOS software checks for the existence of extension ROMs, which contain routines to integrate extra services with the standard ones.

Extension ROMs are identified at 2K blocks between addresses `0xC8000` and `0xE0000`, with `0x55` and `0xAA` in bytes 0 and 1, length in terms of 512 byte blocks in

byte 2 and the code's starting address in byte 3. These ROMs are also checked for the modulo `0x100` checksum over the entire extension ROM length (the checksum should be zero). If these conditions are satisfied, the program in an extension ROM is executed.

After this the `bootstrap` routine in the ROM is executed. This uses the BIOS `0x13` disk services interrupt handler (this handler is referred to as the BIOS `0x13` disk interrupt handler in the rest of this document) to read the `disk bootstrap` routine into memory at a predefined location. The `disk bootstrap` routine is contained in the first sector (the boot sector) of the system startup disk. The `bootstrap` routine in the ROM then transfers control to the `disk bootstrap` routine.

The above steps are common for any PC. Beyond this, the steps become specific to the operating system being loaded.

2.2.2 Loading of PC Operating System

As DOS is the most commonly used operating system on PCs, the loading of DOS will be explained. A more detailed description of the DOS loading sequence can found in [Dun88].

Continuing from earlier, the `disk bootstrap` routine, now in memory, checks to see if the disk contains a copy of MSDOS. It does this by reading the root directory of the system startup disk and determining whether the first two files are `io.sys` and `msdos.sys` (or `ibmio.sys` and `ibmdos.com` for PCDOS), in that order. If these two system files are found, the `disk bootstrap` reads them into memory and executes them to initialize the DOS kernel. In some implementations, the `disk bootstrap` reads only `io.sys` into memory, and `io.sys` in turn loads the `msdos.sys` file. The DOS kernel consists of resident device drivers for console, printer, block devices, etc.

When the DOS kernel has been installed and all resident device drivers are available, normal MSDOS file services are used to read the `config.sys` file. This optional file can contain a variety of commands that enable the user to customize the MSDOS environment. For instance the user can specify additional hardware device drivers known as installable device drivers, the number of disk buffers, the maximum number of files that can be open at one time, etc.

Then the DOS shell or command interpreter is loaded into the memory. The default shell is `command.com`, and a different shell can be specified in the `config.sys` file. Finally, the system startup batch file, `autoexec.bat` is executed to perform additional initializations. The shell then displays a prompt and waits for the user to enter a command. The process of MSDOS booting ends at this point.

2.2.3 Networking and Authentication

It should be noted that up to this point, there has been no provision for authentication of any sort, or for initialization of the network interface. In an open networked environment, unauthenticated users on PCs running DOS put the security of the system at risk, as DOS does not in any way distinguish between ordinary and privileged users. As such, it is advisable to precede the loading of DOS by authentication. The DOS kernel also does not support networking. Various networking software such as PCNFS and XFS can be run on top of DOS to make a distributed file system available to the user of the PC. Prior to the installation of such software, an ethernet adapter card device driver, (e.g. `wd8003e.com` or `ne2000.com`) must be installed on the PC.

2.3 Commercial Background

Many commercially available UNIX implementations (SunOS, HP-UX, etc.) offer a remote boot facility. The availability of UNIX source code has helped programmers understand its bootstrap sequence. As such, the procedure used to boot diskless UNIX workstations is well known. For PCs, Novell Netware and some other commercial vendors provide software for loading DOS remotely from a file server, although little is known about its implementation. However, all the commercial implementations use the common underlying features of a boot ROM and a file server.

2.4 Thesis by Ravichandar

Two M. Tech theses in IIT Kanpur have dealt with the remote boot of PCs. The first, by L. Ravichandar in 1992 [Rav92] adapted the method used to boot diskless UNIX workstations to remote boot PCs.

This was done by first copying the memory contents of a booted PC into a binary file in the order in which they are present in memory. This file is stored on the disk of a remote machine, which acts as the boot server. The boot ROM of the diskless PC uses the File Transfer Protocol (FTP) [PR85] to get this file from the server and load the contents into its own corresponding memory locations. After this, the boot ROM transfers control to DOS.

Such a booting procedure is very different from the step by step sequence of initializations that are followed in the normal DOS boot sequence. As a result, some machine state dependent initializations may be left incomplete, leaving the system in an unstable state. These drawbacks are discussed by Ravichandar.

This approach has not been followed, and as such will not be discussed further.

2.5 Thesis by Manjunath

The second thesis on PC remote boot in IIT Kanpur, by T. J. Manjunath in 1994 [Man94], designed a booting procedure that conformed with the normal DOS boot sequence. This approach was successfully implemented and is described below.

2.5.1 Boot ROM Program Execution

On finding a valid extension ROM at the boot ROM base address of the installed network interface card, the BIOS module transfers control to the program in this boot ROM. The boot ROM code copies itself (using a small routine in its Program Segment Prefix) to the top of the RAM, with 18 kB of space for code, data and stack, and starts execution of the code. After this program has executed, it exits but is left resident in the memory to allow BIOS access to the new `0x13` disk interrupt handler service described below.

2.5.2 Disk Interrupt Handler

The boot ROM contains a new BIOS 0x13 interrupt (disk interrupt) handler to simulate A: drive over the network. The original BIOS 0x13 interrupt vector is stored, and the new interrupt handler is substituted in its place. This new interrupt handler returns default values for all BIOS 0x13 interrupt functions, except for functions 0x02 (disk sector read), 0x03 (disk sector write), and 0x08 (disk parameters). For functions 0x02 and 0x03, it maps the disk address, in terms of track number, head number and sector number, to a file offset. It then accesses the remote boot file at this offset using the protocols and servers described below. For function 0x08, it returns hard coded values for a 1.2 MB floppy disk. It also provides a non standard function 0x1b, which restores the original disk interrupt handler vector in the interrupt vector table (after the new disk interrupt handler's work is finished). The new BIOS 0x13 disk interrupt handler is a central feature around which this remote boot design was based.

2.5.3 Protocols Used

Several standard networking protocols were implemented to support communication between the boot ROM and the UNIX servers.

- **IEEE 802.3:** The client code initializes the Western Digital wd8003e ethernet adapter card at I/O base address 0x280, and obtains the PC's ethernet address from the card. It sets up the card, which sends and receives standard IEEE 802.3 ethernet packets. It operates in a polling mode rather than being interrupt driven.

- **Reverse Address Resolution Protocol:** The PC now knows its ethernet address, but does not know its internet address. This is a generic problem with any machine having no non-volatile storage. So RARP [FMMT84] was implemented to allow the PC to broadcast a RARP packet to resolve its internet address. The reply sent by the RARP server is read by the PC to determine its internet address.

- **Address Resolution Protocol:** ARP as described in [Plu82], was implemented at the client side, both to send queries and to resolve mappings from the internet address to the ethernet address. A small table of mappings was maintained by ARP at the client side.

- **Internet Protocol:** An IP [Pos81a] implementation, was developed upon which the higher layer TCP and UDP protocols could be supported.

- **User Datagram Protocol:** A UDP implementation [Pos80], was developed to contact the UDP server which supplied the initial booting information. Connectionless UDP was chosen over a connection oriented protocol because the initial packet sent to the above server must be broadcast. This is because the server's ethernet and internet addresses cannot be initially known by the client. After receiving the UDP reply, the internet addresses of the both the booting information server and the boot file server are known.

- **Transmission Control Protocol:** A TCP implementation [Pos81b], using the limited features supported by the public domain `tinytcp` software, was chosen to implement the remote boot file access service. This was because the reliability of data transfer was critical and the server's internet address is available at this stage. A full TCP implementation was not used because it would not fit in a 16 kB boot ROM.

2.5.4 Servers Used

Two servers were used, both non-standard UNIX servers.

- **pcbootp Server:** This server was designed to supply the client ROM code with the initial booting information, such as the boot file name, the PC-NFS number and the boot file server internet address (its own address); and also to do acceptance or refusal of password based authentication. It was built on UDP and the requesting PC was identified in the `pcbootp` database, `pcbootinfo`, by the PC's ethernet address.

- **pcboot Server:** This server was designed to provide remote boot file access to the client ROM code. It was built on TCP, and established a new TCP connection with each booting PC. The client code sent a file offset and number of bytes to this server, corresponding to the logical sector on the disk. The client code would open a connection to this server and read the boot sector from the remote boot file before exiting. The connection would finally be closed by the new BIOS `0x13` disk interrupt handler function `0x1b`.

2.5.5 Data Bases Used

Only one data base was used, but it was a non-standard data base.

- **pcbootinfo Database:** This database was accessed by the UDP `pcbootp` server. It would be indexed by the ethernet address of the requesting PC. It stored the name of the boot file that the PC was to use for booting, and the PCNFS number of the PC.

2.5.6 RAMDrive Usage

Another important requirement of this remote boot procedure is the use of a RAM-Drive on which to duplicate the contents of the remote boot file. The reason is intricate. The ethernet adapter card device driver developed provides a skeletal and non-standard network interface. So it must be replaced by a standard ethernet adapter card device driver (e.g. `wd8003e.com`) before XFS or PCNFS is installed. However, doing this will immediately disable the remote A: drive, after which the files required to install XFS or PCNFS are no longer accessible from the remote boot file being simulated on A: drive. Therefore, before the standard ethernet card device driver is installed, all the files still required for booting must be made available locally on the PC. But the PC may be diskless. So the only solution is to simulate a disk in the RAM, into which the necessary files can be copied and later accessed. This simulated disk is called a RAMDrive. A new RAMDrive creation utility, called `ramdisk`, had been written, as DOS's RAMDrive utility cannot later release its memory.

2.5.7 Bootfile Restrictions

Three restrictions had been defined on the boot file.

- **RAMDrive Location:** Some lines need to be specified in `config.sys`, which ensured that the RAMDrive would be created with a predefined drive name, so that it could be correctly accessed.
- **Call to ramdisk:** A call to `ramdisk` must be present in the batch files run during booting. This call must be before access to the RAMDrive. The `ramdisk` program must be present on the boot file.
- **Call to reset:** A call to `reset` must be present in the batch files run during booting, and the `reset` utility must be on the boot file. The call to `reset` must be made after duplicating the remote boot file on the RAMDrive. This call must made be before the installation of the standard ethernet card device driver, i.e. `wd8003e.com`. The `reset` utility calls the new BIOS `0x13` disk interrupt handler function `0x1b`, to restore the old BIOS `0x13` disk interrupt handler.

2.6 Starting Point for the Thesis

The sections described in this chapter describe the earlier framework and the starting point for this thesis. The problems in the earlier implementation are summarized below.

- PCs' disks not operational
- No boot management flexibility
- Cryptic, non-standard server database
- File server unreliable
- No password security

- No spare ROM code space
- Hard coding of variable disk parameters
- Support for single ethernet card & I/O base address
- Code quality
- Documentation quality

As a result of these problems, the earlier remote boot system could not be successfully installed in IIT Kanpur's Computer Center.

Due to the number and complexity of the design and implementation changes that had to be made, it was necessary to completely re-design PC remote boot, rather than just modify the earlier code. Nonetheless, the work done by T. J. Manjunath serves as the starting point for the work done in this thesis.

Chapter 3

Design Issues

This chapter describes the design considerations that were taken into account to develop a high quality remote boot product. Some of these were limitations which were not to be violated under any case, and others were specifications which were less stringent but equally important. A few of the limitations were carried over from the earlier implementation, but have been explained here for clarity. The remaining issues either deal with drawbacks in the earlier system or with more general modifications. An overview of the implementation has been given at the end of this chapter, and the implementation is dealt with in detail in the following three chapters.

Understanding of the design issues is necessary for understanding the goals of this thesis, and for understanding the complexity of the re-design problem handled. The issues are separated into server and client design limitations, and server and client design specifications. All issues are explained in terms of their background, so that they are easier to understand.

3.1 The Need for a New Design

In chapter 2, the earlier implementation on which this thesis is based was explained. This implementation resulted in a working remote boot system, but one which had several minor and major limitations and problems in its usage. Not all of them

were apparent at the time this thesis was begun. Many were identified during the design stage of the new remote boot product, and some were also identified during the testing of the created product.

The complexity of the design issues discussed below makes it clear why there was a need for a new design for the remote boot system.

3.2 Design Limitations: Server

3.2.1 TCP/IP Supported Protocols

This limitation seems obvious, as most commonly developed networking applications are based on the TCP/IP protocol suite. However, it it should be noted that there are networking and transport layer protocols in use, e.g. SNA, DECnet, etc., which are not based on TCP/IP. As such, this server limitation means that the remote boot system developed is limited in use to networks which provide support for TCP/IP.

3.3 Design Limitations: Client

3.3.1 Upper Limit on ROM Code Size

ROM BIOS has provision to support extension ROMs up to 128 kB in size (i.e. 256 blocks, each of 512 bytes). However, standard network interface cards only support limited boot ROMs of various sizes (like 8 kB, 16 kB or 32 kB), as this has been found sufficient for boot ROM applications. If several extension ROMs are being used, they will compete for address space in the memory [Gill94]. Therefore the minimum possible boot ROM size should be used. As cards supporting 8 kB boot ROMs are relatively less common, and as 8 kB boot ROMs would be infeasible for PC remote boot, a strict upper limit of 16 kB was adhered to for the boot ROM code size.

3.3.2 Standard Boot Interface

When the user has commenced the booting of a PC, there is a standard sequence of operations which occur, and which the user expects. Any significant deviation from this standard behavior may make the user uncomfortable with the system. Thus, apart from authentication, there should be no or minimum change in the user interface of a remote boot from the user interface of a local boot.

3.3.3 No Usage of DOS Functions

When the boot ROM in the network interface card takes over control from the ROM BIOS, DOS operating system calls are not available because DOS has not been loaded and installed. (Actually, no operating system is available). As such, while programming the client code, usage of DOS functions will result in fatal run-time errors, and so their usage must be avoided.

3.3.4 No Changes in PC other than Boot ROM

PC's come with standard built-in hardware and ROM-BIOS software. Although it is possible to make unsupported modifications in these in order to support remote boot, this would be an erroneous approach. This is because future changes made by PC vendors may make such modifications ineffective or conflicting, thus making the remote boot product obsolete. As such, only standard supported modifications, such as the addition of a boot ROM, are to be allowed.

3.3.5 No Degradation in Performance

Performance is a critical issue wherever a user has to wait for an application to complete its work. Booting is one such issue. The redesign of remote boot, to meet the required limitations and specifications, should not result in a noticeable degradation in the time taken to boot. On a 10 Mbs LAN, network access is faster than floppy disk access, but slower than hard disk access. In the worst case, remote boot should not be any slower than a floppy disk boot.

3.4 Design Specifications: Server

3.4.1 Providing Boot Management Flexibility

The remote boot procedure and its features may need to be configured in a simple and centralized manner by the system administrator. It may be required on some PCs to allow a local disk boot, or to bypass authentication, or to boot with a new boot file, or to allow the booting process to modify the boot file, or to test with a different server, or to print debugging information at the PC, etc. Thus, a specification is to provide to the system administrator simple-to-use boot management flexibility for the above features, in a single, centralized and secure place.

3.4.2 Usage of Standard Services

There are a number of services, or protocols, which are defined as standard protocols [Com88] by the Internet Activities Board (IAB). All hosts and gateways are required or recommended to implement these services. As such, they come freely available with any UNIX implementation. If the remote boot process can be constrained to use only such services, there will be no need of any explicit remote boot server. This will make the remote boot system portable to any network without the addition of any new services, which is very attractive from an acceptability point of view. Thus, a specification is to use a standard service wherever there exists a standard service which can perform the required job.

3.4.3 Minimization of Non-Standard Services

There may be some jobs for which standard services do not exist or for which the use of the standard services will result in violations of the design limitations such as the boot ROM code size. As such, some non-standard services may have to be used. The more non-standard services that are used, the more numerous will be the possible points of remote boot failure, and thus the reliability of the remote boot system will decrease. Thus, a specification is to minimize (by merging or elimination) the number of non-standard services that are being used for remote boot.

3.4.4 Elimination of Non-Standard Data Bases

Although there may exist some non standard services, is was decided to disallow the existence of non standard databases. This is because a reliable non-standard service will need little or no maintenance by the system administrator, whereas a non-standard database will require regular maintenance whenever changes are made in the system configuration. The addition of a non-standard database will thus make a new product unattractive for system administrators. As there are already a variety of configurable standard databases available for standard services on UNIX implementations, a specification was to disallow the use of non-standard databases altogether.

3.4.5 Ensuring Portability Across Servers

The PC's 80x86 architecture has been designed to ensure portability of data types across different generations of machines. There is no such compatibility amongst the various machines – supplied by different vendors – that can play the role of a UNIX server in a remote boot system. For example, an *int* data type may be 16 bit or 32 bit, and if this is not taken care of, server code written and tested on one machine may not work on a different machine – a major drawback. Thus, a specification was to develop fully portable server code.

3.4.6 Logging of PC Remote Boot Activities

There are a number of reasons why a history of PC logins should be maintained. PCs running simple DOS constitute a security risk in an open networked environment, as the PC user has all system privileges, and may act maliciously. For these reasons, and in order to estimate PC usage and availability for planning purposes, it is useful to maintain a log of PC login activity which is available to the system administrator. Thus, a specification was to maintain a standard, secure and easily readable database to record login activity on PCs.

3.5 Design Specifications: Client

3.5.1 Usage of Low Weight Protocols: UDP

The weight of a protocol refers to its overhead in terms of packet header length, initialization time, packet processing time and implementation code size. A heavy weight protocol (e.g. TCP) generally results in an increase of reliability and bulk data transfer speed, at the cost of all the above factors. However, for the remote boot application in which 600 byte packets have to be transferred over a local LAN, light weight protocols (e.g. UDP) are not at a disadvantage in terms of reliability or data transfer speed. The advantages of using light weight protocols then become primary. Thus, a specification was to use only light weight protocols.

3.5.2 Optimization of Code, Data and Stack Size

The optimization of client code, data and stack size is a different issue than the upper limit on the ROM code size discussed earlier. By further reducing the ROM code size below the 16 kB limit defined, more functionality (e.g. extra ethernet card drivers) can be added, more meaningful diagnostics can be provided, and space can be left to accommodate code in the future due to maintenance and upgradation. However, such omissions should not take place at the cost of the quality of the remote boot. The boot ROM program is copied to the top of the PC's 640 kB RAM, with sufficient space for its data and its stack. As this program must be left resident in memory when DOS is initialized, the memory used by remote boot at the top of the RAM is never available to DOS later. Thus, optimization of the code, data and stack size also makes more memory available to DOS.

3.5.3 Transparent Handling of Disk Configurations

The boot file that stores the binary image of a floppy disk contains parameters which are specific to a particular floppy disk. When these are accessed during booting, they are stored in the local BIOS disk parameters table [Gill94]. If the local PC supports a different A: drive floppy type than that of the boot file, A: drive may

not be accessible to the user later. In addition, the number of drives seen by the remote boot system will affect the later availability of both the hard disks and floppy disks. Disk unavailability will not be acceptable to users. Thus, a specification was to make remote boot handle any disk configuration transparently, i.e. all disks supported should be available after booting. Further, a diskless PC should also be able to boot and run.

3.5.4 Operating System Independence

Today, DOS and Windows are not the only operatings system being used on PCs. Many users want their PC to provide them with the secure and, in some cases, the familiar environment of another operating system such as UNIX, Linux, OS/2, NetBSD, or Mach. The ROM BIOS interface used at initialization time by all these operating systems is necessarily the same. However, some things such as the boot floppy format type, may vary between operating systems. Any hard coding specifically for DOS may make other operating systems unloadable by remote boot. Thus, a specification was to design the remote boot to make it transparent to the operating system being loaded.

3.5.5 Providing Multiple Ethernet Card Support

There are two ethernet adapter cards that are most commonly used in PCs, the `ne2000` (or its compatible), and the `wd8003e` (or its compatible). In addition the I/O base addresses of these cards can be at various locations. If only one card or one I/O address is supported by remote boot, it can only be used in a very limited set of networks, or its usage would require changes in the networked PCs' hardware. Thus, a specification was to identify and support both these cards at any of their recommended I/O addresses, with the use of the same boot ROM.

3.5.6 Handling of Security Issues

When the user enters a login name and password, these are stored by the remote boot program at the top of the memory, in data structures used by protocols from

the application to the MAC layer. As this memory is not reclaimed by DOS, it remains unmodified, and can be scanned for copies of the user's password. Another issue is the transmission of the unencrypted password across the LAN. Any network snooping utility such as `tcpdump` can be used to pick up the login packets and then scan them for passwords. Thus, a specification was to eliminate such loopholes in security.

3.5.7 Providing Improved Diagnostics

If for some reason the remote boot system does not work, it must be made re-operational as quickly as possible. This is impossible if the diagnostics reported at the PC and at the server are absent, excessively brief, misleading, badly laid out or cryptic. Clear and concise diagnostics make the system both pleasant to use and easy to maintain. However, diagnostics may use up a lot of scarce ROM code space. Thus, a specification was to use improved and intelligent diagnostics.

3.5.8 Provision for Testing without a Boot ROM

The cycle of making changes in the boot ROM code, creating a file to download to a ROM, programming the ROM, installing it on a PC, testing it and then erasing the ROM for its future reuse is a lengthy cycle. A Terminate and Stay Resident executable (TSR) was earlier implemented which tests the ROM code as a user program, thus avoiding the above cycle. With all the issues involved in the re-design, this TSR program must be updated to test the functionality, and not have dependencies on old functionality that was removed. Thus, a specification was that the new remote boot system should also be testable in the TSR mode without the programming of a boot ROM.

3.6 Design Specifications: Software Quality

3.6.1 Producing High Quality Code

Code quality is a critical issue if a software product is to have a long lifetime. In the earlier implementation, this was noticeably absent, and a third of the time spent on this project was spent on fully understanding the earlier code. Following a good set of guidelines while writing code is essential if the code may be later read by someone else. Thus, a specification was to choose and adhere to a standard set of software programming guidelines, resulting in high quality code.

3.6.2 Providing Complete Documentation

Even a simple software system needs comprehensive documentation of its features in order that it may be used correctly. When a system consists of complex and multi-platform software, databases and floppy image files, hardware modules and processes and hardware programming and erasing utilities, the system may become unimplementable due to its complexity. As such, a specification was to provide step by step procedures for all the stages involved in implementing the remote boot system, together with any trouble shooting information that would be useful.

3.7 Overview of the Implementation

The steps followed to implement PC remote boot are summarized below. They are explained in more detail in chapters 4 – 6. The sequence of steps listed starts from the point where the ROM BIOS hands over control to the boot ROM.

- (1) The `act.bin` binary (a program lying between bytes 0 and 255) in the boot ROM copies `main.com` to the top of the RAM. It reserves 18 kB for `main.com`'s code data and stack, changes the segment registers and then hands over control to `main.com`.

- **(2)** The boot ROM probes for the presence of `wd8003e` or `ne2000` ethernet adapter cards. If either is present its name and location are displayed, else an error is reported and booting stops.

- **(3)** The ethernet adapter card is initialized and its ethernet address is read and displayed.

- **(4)** The PC formats and broadcasts a BOOTP request packet on the BOOTP server port number. The PC then polls for a BOOTP reply, and sends a fresh packet every 2 seconds if a reply is not received.

- **(5)** The `inetd` daemon process, on receiving the BOOTP request packet, starts the BOOTP server if it is not already running. This server extracts the client's ethernet address from the packet and uses it to find the client's entry in the `/etc/bootptab` file. On finding an error free entry, a BOOTP reply packet is formatted and sent to the client PC. Errors are logged in `daemon.log`.

- **(6)** The client reads the BOOTP reply packet. This contains boot configuration control information, the client's IP address and the boot file name. The client IP address and boot file name are displayed.

- **(7)** If authentication is to be done, the PC prompts users for their login and password. These are encrypted and broadcast on the authentication and boot file server port number. The PC then polls for an authentication reply, and sends a fresh packet every 2 seconds if a reply is not received.

- **(8)** The `inetd` daemon process, on receiving the authentication request packet, starts the authentication and boot file server if it is not already running. This server reads the client's authentication request, decrypts the login and password, and accesses the NIS password database entries to check the login–password combination. It sends an appropriate authentication reply. The request is logged in `daemon.log`.

- **(9)** The client reads the authentication reply packet. If authentication failed the client goes back to step (7).

- **(10)** If a local boot is to be done, this is reported to the user and the boot ROM code exits. Booting then proceeds from the PC's local disks.

- **(11)** The PC prompts the user to enter a different boot file name. If no name is entered, the boot file name sent by the BOOTP server is taken as the boot file.

- **(12)** The PC sends a read request packet to the authentication and boot file server to read the boot sector of the boot file. The file offset is calculated from the sector, head and track number by using the following formula:

```
Logical sector  =  (cylinder * number of sectors per cylinder)
                   + (head * number of sectors in a side)
                   + (sector - 1);

File offset     =  logical sector * 512;
```

  After any read / write request is sent to this server, the PC polls for a read / write reply, and sends a fresh packet every 2 seconds if a reply is not received.

- **(13)** The authentication and boot file server reads the client's read / write request, and reads / writes 512 bytes of the boot file at the requested offset. A read / write reply is formatted and send back to the client.

- **(14)** The PC extracts the boot sector parameters from the reply packet and stores them. These are used to calculate the number of cylinders and the remote boot drive type, and to check for invalid boot files.

- **(15)** The PC stores the old floppy disk parameters using the `0x08` function of the old BIOS `0x13` disk interrupt handler.

- **(16)** The PC stores the old BIOS `0x13` disk interrupt handler vector. It then changes this vector to point to the new remote disk interrupt handler. Now any read / write requests to A: drive will result in accesses to the remote boot file, as explained in steps (12) and (13).

- **(18)** If the *TEST* flag was specified during compilation, the PC tests the new BIOS `0x13` disk interrupt handler. It then calls *geninterrupt(0x19)* to bootstrap the operating system from the remote boot file, and then terminates and stays resident in the memory.

- **(19)** The boot ROM routine exits and returns control to the ROM-BIOS.

- **(20)** ROM BIOS bootstraps the operating system from its A: drive. As the BIOS `0x13` disk interrupt vector has been changed, this results in the contents of the remote boot file being loaded as the operating system.

- **(21)** During booting of MSDOS, a RAMDrive is created in the PC. Boot file contents are copied to this RAMDrive. Then the new BIOS `0x13` disk interrupt handler function `0x1b` is called. This resets the BIOS `0x13` interrupt vector back to the old disk interrupt vector stored in step (16). The remote boot file is now no longer accessed.

- **(22)** The authentication and boot file server times out after 1 minute and exits. The BOOTP server also times out after 15 minutes and exits.

- **(23)** The RAMDrive is identified and the remaining parts of the the boot (e.g. installing the permanent packet drivers, starting XFS, etc.) use the files stored in the RAMDrive.

- **(24)** The PC is now fully booted, and its behavior is the same as that of a PC booted from its local disks.

Chapter 4

Server Implementation Details

The last chapter dealt with the design limitations and specifications. The implementation of a design is a significantly different issue. Given a set of requirements for a design, there can be a number of different ways to implement them. The one to be chosen is dependent on many factors such as cost, availability, etc, and these choices form the implementation details.

In this chapter the details of the implementation of the remote boot server are explained, and the choices made for the implementation of each of the design specifications are justified. Although these details are primarily concerned with the server implementation, the corresponding details of the client interaction are also explained.

4.1 Usage of BOOTP Service

The BOOTP protocol was used to implement the server which is to provide the initial boot configuration information to the PC. This is one of the key aspects of this remote boot implementation. In this section, the unelaborated terms server, client and packet, refer to the BOOTP server, client and packet respectively. Various aspects behind the choice of BOOTP are also explained below.

4.1.1 History of Development

The BOOTP protocol is a standard protocol recommended by the Internet Activities Board. It is extensively used for the booting of diskless workstations on a network, and as such is available on most UNIX or UNIX like implementations. It has a long and stable history of development, as given in [CG85], [Prin88], [Rey88], [Wim93] and [AD93]. The version of BOOTP supported by the server is defined by the Request For Comments (RFC) number, corresponding to which there is a *magic cookie* sent in the server's packet. The service defined by `rfc1048` is upward compatible with the service defined by `rfc1533`, but the reverse is not true. As such, the client has been designed to ignore the magic cookie, and to expect `rfc1533` service. This approach enables the client to interpret packets sent by `rfc1048` servers, `rfc1533` servers, and servers implementing any future versions of BOOTP which are compatible with the `rfc1533` service.

4.1.2 Details of Use

When the ROM code has identified and initialized its ethernet adapter card interface, it can send and receive packets. However, as the PC may not have any non-volatile storage, it cannot be assumed to know its internet address, and only knows its ethernet address. So the PC uses a broadcast address to communicate [Mor84]. In this broadcast packet, the client ethernet address is the true address available in the network interface card, the client internet address is null, and the server ethernet and internet addresses are broadcast addresses. The broadcast is sent from the client UDP port 68 to the server UDP port 67, both being reserved for use by BOOTP.

In the request packet created by the client, its ethernet address is filled for matching by the BOOTP server in its configuration file, `/etc/bootptab`. A random identification number is also sent, so that the received reply can be matched with the sent request. The number of hops is set to zero, so that only a server on the local LAN is contacted. Other fields are left null, and the packet is transmitted over the LAN.

The server can be started either as an independent process, or as an `inetd` daemon process. The former can be used for testing, and the latter should be used

for the permanent service. These details are available in section A.3.2. The server picks up the packet and matches the sent ethernet address with the host ethernet addresses in the `bootptab` configuration file. Optionally it logs the receipt of the request packet and any error messages caused by incorrect syntax in the `bootptab` file, in the `daemon.log` file. This is done by specifying the appropriate debug level in the command to start the server. This option is useful for debugging and for maintaining a record of PC login activity.

A number of fields are obtained from the `bootptab` file or other databases such as Network Information Services (NIS). These are packed into the reply packet and sent back to the client. In the reply packet, some of the fields are essential, while some others are optional and others are default. Essential fields include the boot file name, physical layer type and address length. Remote boot does not need the subnet mask. Default fields include the IP address of the client. Optional fields include the internet address and port number of the authentication and boot file server, the client hostname, and a boot configuration control flag.

These fields are placed either in standard locations in the reply packet, or as a sequence of bytes with tag and length identifiers within a 64 byte array known as *vendor extensions*. The client's internet address and the matching identification number are always present in standard locations in the reply. The vendor extensions may have standard tags (from 0 to 127, 255), or they may have implementation defined tags (128 - 254). This remote boot implementation uses the new and optional tags 128, 129 and 130 to store the boot configuration control flag, the authentication & boot file service port number and the authentication & boot file server's IP address respectively. Although there is a standard tag for the boot file server's internet address, it is not used. This is because it is defined by BOOTP as the address of a TFTP service, and our boot file service does not implement TFTP. The reply packet is sent to the client on port 68, using the client's ethernet address and a broadcast internet address (as the client does not know its IP address yet).

4.1.3 bootptab File

The information present in a valid `bootptab` host or client entry includes the client name, the physical layer type, the physical address length, and the physical layer address, all of which are essential. Other information includes the boot file name, the boot configuration control flag, the subnet mask, and the port number & internet address of the authentication & boot file server, all of which are optional. If essential information is not found in the `bootptab` file, or if information from other databases (such as the client's internet address) cannot be obtained, or if a syntax error is found in the `bootptab` file, the server logs an error in `daemon.log` and does not send a reply.

The boot configuration control flag is a bit-wise flag with options for remote or local boot, authentication or no authentication, control of write permission to the boot file, and step by step printing of debug information when the client is complied as a TSR program. The client defaults are remote boot, authentication, no write permission and no printing of information. All these options are simple to control from the `bootptab` configuration file.

The subnet mask is set to 255.255.255.255 by default if not specified, but is not used by the remote boot. Some other programs like XFS, etc., use this field after contacting the BOOTP server at a later stage. By using the subnet mask, the first packet sent to the authentication and boot file server becomes a subnet broadcast rather than a full LAN broadcast.

The port number and IP address of the authentication and boot file server are meant for testing a client with a new authentication and boot file server, without disturbing the existing service. The default for the authentication and boot file service port number is 146. This choice was made as the service must be secure from user impersonation and therefore must lie in the port number range 0 – 1023, and as this port number must be unused by any standard UDP service (this was verified from the `/etc/services` system file). The default for this server's internet address is the broadcast address. This is possible because the use of UDP for the boot file service in place of TCP allows communication without knowing the server's internet address. The authentication and boot file server's correct internet address

is used after receiving the first reply from this server.

4.1.4 Advantages of Use

- BOOTP protocol is built on top of UDP, and so it can be supported on all networks which support at least UDP over IP.

- Boot management flexibility is provided in a single centralized and secure place. The options in the `/etc/bootptab` file can allow the system administrator to decide for each PC on the network whether it should boot from local disks or do a remote boot; whether authentication is required or not; which boot file should be used for boot; which authentication & boot file server is to be used (in terms of IP address and port number); and whether the boot process can modify the boot file. In addition, it allows printing of debugging messages with the run of a Terminate and Stay Resident version of the client program.

- The BOOTP protocol is a standard protocol as defined by the Internet Activities Board. All hosts and gateways supporting UNIX are recommended to implement the BOOTP protocol. Therefore the remote boot system's usage of the BOOTP service does not decrease its portability to new networks (given that they have UNIX servers).

- The usage of the BOOTP protocol helps avoid the use of the `pcbootp` server used in the earlier implementation of remote boot. This reduces the number of non-standard services that are used on the server side, and simplifies the server implementation. This increasing the reliability of the remote boot system.

- The usage of the BOOTP protocol eliminates the use of the `pcbootinfo` database used in the earlier implementation, and replaces it with additional entries in the standard `/etc/bootptab` configuration file. As the `bootptab` file is already used to configure the boot of diskless workstations, using it to configure the boot of diskless PCs does not change the semantics of its usage. This makes the maintenance job easy for system administrators. Thus, the

use of non standard databases is completely eliminated in the remote boot system.

- The BOOTP service has built in features to allow it to log request packets, reply packets and errors in the bootptab file into the daemon.log file. This maintains a part of the required remote boot usage history, and makes the identification of syntactical errors in the bootptab file easy.

4.1.5 An Example Configuration File

Sections of a standard bootptab file modified to support PC remote boot are shown below. The comments make it self explanatory.

```
# Example /etc/bootptab: database for bootp server (/etc/bootpd).
#
# Format:
#       nodename:tag=value:tag=value: ... :tag=value

###############################################################################
############################## PCBOOT OPTIONS ##################################
###############################################################################

#   Optional vendor specific flags for PC boot are explained below
#   T128=C0:\
#   b7: 0 => Local boot,          1 => Remote A: drive boot          DEFAULT: 1
#   b6: 0 => No authentication,   1 => Authentication required       DEFAULT: 1
#   b5: 0 => Remote A: drive      1 => Remote A: drive writable      DEFAULT: 0
#            never writable            before reset of disk handler
#   b4: 0 => No debug info        1 => Print debug info with Term.   DEFAULT: 0
#                                      & Stay Resident version
#   b3-b0 => Not used                                                DEFAULT: 0
#   T129=0092:\
#   Authentication + boot file server port number                    DEFAULT 0x0092
```

```
#   T130=9010A221:\
#   Authentication + boot file server IP address   DEFAULT  subnet broadcast
#   NOTE: After getting the first valid reply,
#         the correct IP addr is set

.PC_DEFAULTS:\
        ht=ethernet:hn:sm=255.255.0.0:vm=rfc1048:\
#       ht: Ethernet LAN, having 8 byte hardware addresses
#       hn: Write host name in the BOOTP reply
#       sm: Subnet mask is 255.255.0.0
#       vm: BOOTP version corresponding to rfc1048
#       NOTE: hosts internet address is picked up from the NIS
#       T129=0092:\
#       T130=9010A221:\

# PCs with wd8003e cards must install the wd8003e packet driver
pc10:   tc=.PC_DEFAULTS:ha=00803c570040:T128=C0:bf="/usr/adm/xfswd.img":
pc11:   tc=.PC_DEFAULTS:ha=0000c00f00dd:T128=C0:bf="/usr/adm/xfswd.img":
pc20:   tc=.PC_DEFAULTS:ha=0000c00f007a:T128=C0:bf="/usr/adm/xfswd.img":
pc35:   tc=.PC_DEFAULTS:ha=0000c054002b:T128=C0:bf="/usr/adm/xfswd.img":
pc37:   tc=.PC_DEFAULTS:ha=00803c57003a:T128=C0:bf="/usr/adm/xfswd.img":

# PCs with ne2000 cards must install the ne2000 packet driver
pc12:   tc=.PC_DEFAULTS:ha=00803c57002f:T128=C0:bf="/usr/adm/xfsne.img":
pc52:   tc=.PC_DEFAULTS:ha=0000E8C3A59b:T128=C0:bf="/usr/adm/xfsne.img":
pc59:   tc=.PC_DEFAULTS:ha=0000e8c51bdc:T128=C0:bf="/usr/adm/xfsne.img":
##############################################################################
############################  PCBOOT OPTIONS  ################################
##############################################################################
```

4.2 Single Authentication and Boot File Server

The earlier implementation used two non-standard remote boot services, the UDP based `pcbootp` for the initial booting information and authentication, and the TCP based `pcboot` for access to the boot file. The initial boot information which was supplied by `pcbootp` is now supplied by the BOOTP service, along with other information. The two remaining tasks are authentication and boot file access.

To do these tasks, the `pcbootp` and `pcboot` servers may be used. However, this would increase the complexity of the system and reduce its reliability. So a new `pcboot` server was written, which would supply both authentication and boot file access services in a single service.

Before the development of a new service, it was necessary to see whether there exist standard protocols which can perform the required tasks. This is because by the elimination of all non-standard services, a much more portable and reliable remote boot product could be developed. However, this was not possible for the reasons listed below.

- For authentication, there is no standard protocol recommended by the Internet Activities Board. The closest match is a TCP authentication service auth (port 113), as discussed in [StJ85]. However this provides a very limited authentication mechanism based on port numbers for a specific TCP connection, and it is not a more generic password authentication service to authenticate a user. There are two UDP authentication services, the `passwd_server` (port 752) of `kerberos`, and `rauth2` (port 2001). However, neither of these are standard protocols, and so need not be considered as they will have to be rewritten to make the remote boot system portable.

- The BOOTP protocol assumes that TFTP will be used for access and transfer of the remote boot file [Fin84]. This is possible on workstations, where the boot file is transferred as a single in-order file before control is transferred to it. While booting DOS or a general operating system, this assumption does not hold. DOS sees the boot file as a floppy disk on A: drive, and accesses the floppy's sectors in any order. TFTP can only transfer consecutive blocks

of a file [SCS1], and so cannot be used for the remote boot of PCs. The File Transfer Protocol, [PR85], has the same problem.

- Another option is to use the Network File System (NFS) protocol [Sun89] which is built on top of SUN's Remote Procedure Call (RPC) protocol [Sun88]. Sun RPC is built up using SUN's eXternal Data Representation (XDR) [Sun87]. This meets our requirements of a file access protocol which is both a standard protocol and can give any order file access. However, at the time of the design of the new remote boot system, it was felt that this approach would use up too much scarce ROM code size, as it would require the implementation of 3 new protocols, 2 of which (NFS and RPC), would have to support complex features. As such, this option was also abandoned.

For these reasons, a new `pcboot` service was written, which would perform both authentication and boot file access. New data structures were developed which allow a single packet format to be used for both the services, both for requests and for replies. The operation code in the requests and replies made the nature of the packet clear, i.e. for authentication or boot file service. Such an approach simplified the overall system design.

Key packets relating with this new `pcboot` service, such as those indicating boot file access failure, and all packets relating with authentication, are logged in the `daemon.log` database, which is a standard and secure database. This allows the system administrators to diagnose remote boot failure quickly and also keep track of PC logins by users.

It should be noted that both of the non-standard protocols used in the earlier implementation could not be eliminated, as there existed no standard authentication protocol for user authentication. By reducing the number of non-standard protocols used from two to one, the server side reliability is enhanced.

4.3 Ensuring Portability Across Servers

4.3.1 Usage of Platform Independent Data Types

In order to handle the portability of server code across UNIX platforms supporting different machine architectures, a set of data types defined by Dr. Moona were used for the implementation of all data types used in the authentication and boot file server. These data types are listed below.

```
/******************** Unsigned Data Type Definitions ************************
typedef unsigned int        ubyte4;   /* 32 bit unsigned */
typedef unsigned short      ubyte2;   /* 16 bit unsigned */
typedef unsigned char       ubyte1;   /*  8 bit unsigned */
/******************** Unsigned Data Type Definitions ************************

/********************** Signed Data Type Definitions ************************
typedef int                 byte4;    /* 32 bit signed */
typedef short               byte2;    /* 16 bit signed */
typedef char                byte1;    /*  8 bit signed */
/********************** Signed Data Type Definitions ************************
```

By avoiding the use of any other data types in the server code than those defined above, it was ensured that all the server code is portable to any UNIX platform. Portability across machines is achieved by changing the above definitions only.

Chapter 5

Client Implementation Details

Chapter 4 dealt with the remote boot server side implementation details. In this chapter the details of the implementation on the client side are explained, and the choices made for the implementation of each of the design specifications are justified. Again, although these details deal primarily with the client implementation, the corresponding details of the server interaction are also explained.

5.1 Handling the BOOTP Information

The server side issues relating to the BOOTP protocol were dealt with in the chapter 4. The client aspect, that is how the boot information in a successfully received reply packet is interpreted and used to handle the flow of control in the client code, is dealt with here.

On the successful receipt of a BOOTP reply packet at the client, it extracts information from the packet. The server does not send an invalid reply, as was explained in the chapter 4. Further, as the optional fields have built-in and valid defaults in the client code, there is no possibility of incomplete or incorrect information being present with the client once a valid BOOTP reply has been received. For a reply to be recognized as valid, the sent and received identification numbers must be the same. (Note: the boot file name may be incorrect, but this cannot be known at this stage). However, if the client does not receive a valid reply, it cannot proceed

with the remote boot – the client then continues to try to contact the server in an infinite loop, reporting after every 16 seconds that the BOOTP server could not be contacted. Under this case, it should not proceed with a local boot, as this would become a loophole in the system security.

On receiving a valid reply, it is interpreted as follows. The client's IP address in the reply packet is stored and used in all future communication by the client. The boot file name is read and stored, but for testing purposes and for boot flexibility at the client side, the user is allowed to override the boot file choice with a new choice. The client hostname is printed for the PC to be identified before booting.

The boot control flag is a byte wide vendor extension, consisting of 8 bit-wise flags. Bit 6 is used to define whether the PC boot requires an initial authentication (= 1) or not (= 0). If required, the authentication routine is called, which executes in an infinite loop until a correct login-password combination is entered.

Bit 7 is used to define whether the PC boot should be from local disks (= 0) or remote boot file (= 1). If a remote boot is to be done, the client code changes the BIOS `0x13` disk interrupt handler vector in the interrupt vector table to the new remote disk interrupt handler, and then the client program exits. If a local boot is to be done, the interrupt handler vector is left unchanged, and the client program exits.

It should be noted that the functionality of bits 6 and 7 are independent, and any combination of local vs. remote boot and authentication vs. no authentication can be specified for a PC. The other bits that are used in the boot control flag, i.e. bits 4 and 5, do not significantly affect the flow of control, and the `bootptab` example given in chapter 4 should be referred to to understand them. Bits 0 – 3 are not used at all, and should be zero to be compatible with future extensions.

5.2 Usage of UDP for Boot File Access

In the earlier implementation, the Transmission Control Protocol (TCP) [Pos81b] was used to develop the boot file access server, `pcboot`. This was replaced by the use of the User Datagram Protocol (UDP) [Pos80] to implement the authentication

and boot file access sever.

The reason for doing this was that UDP being a light weight protocol, its use results in far fewer overheads in processing, initialization, termination and client ROM code size than by using the heavy weight protocol TCP. This gives advantages in terms of booting time and code size, as shown in the results given in chapter 8.

The fact that all sectors of a disk are of 512 bytes means that any sector's data can be put into a single UDP packet. Thus, a request to the new BIOS `0x13` disk interrupt handler to access `N` consecutive sectors can be broken down into `N` separate accesses to the remote boot file. The semantics of the operation are fully maintained. Further, on a LAN, the reliability of TCP and checksum based UDP are effectively equivalent for our remote boot system. These were the key factors which allowed UDP to be used for boot file access from the client side.

Apart from this, the use of TCP can result in half open connections if the PC is rebooted before the existing TCP boot file server connection is terminated. On the Dec Alphas used in our environment, these half open connections and their corresponding server processes remained existing for long periods, slowing down the system.

The use of UDP meant that a single server could be used for both authentication and boot file access, thus simplifying the server side of the remote boot system as discussed in chapter 4.

5.3 Optimizations of Code, Data and Stack Size

This was one of the most important goals of this thesis work, as substantial functionality was to be incorporated without exceeding the defined 16 kB ROM code size limit. Optimization of the RAM code, data and stack size was also to be done. Many approaches were used, some of which were quite innovative.

5.3.1 Elimination of RARP, ARP and TCP

The code implementing RARP, ARP and TCP was removed from the client side of the remote boot system, freeing about 6 kB of the 16 kB ROM code size. The

reasons why this was possible are described below.

- Because the BOOTP protocol does not require the client request packets to contain their IP addresses, the client does not need to know its IP address before it can contact a server. Earlier, the Reverse Address Resolution Protocol, [FMMT84], was used to get the client's IP address from its ethernet address, by taking the help of RARP servers on the network. Thus, using BOOTP allowed the client code implementing RARP to be removed.

- The implementation of the Address Resolution Protocol (ARP), [Plu82], seems to have been an oversight in the earlier implementation. It is true that the booting PC receives ARP requests to supply its ethernet address given its internet address. However, such requests can be ignored at no significant cost until the PC's booting process is complete and its operating system is installed. As such, the client code for ARP could be removed.

- RARP and ARP are both light weight protocols, and the code size required to implement them is small compared with the code size required to implement a heavy weight protocol such as TCP. This was a major reason why TCP was bypassed by using UDP to develop the boot file access service. The elimination of TCP saved about 5 kB of ROM code size.

5.3.2 Efficient Global Variable Declarations

There were two aspects to this optimization, the obvious precaution of avoiding the duplication of global variables, and the less obvious step of centralizing the declaration of all global variables at one place.

- Large data structures may be used by multiple routines. An intelligent design of the routines that use them means that a single data structure can be used by all the routines without any conflicts. This was done for all large data structures, saving considerable space.

- By declaring all the global variables at a single place, e.g. the `main.c` file, an interesting aspect of the behavior of the Turbo C linker was observed: code

size equal to the size of the data structure is saved. For example, if the 1518 byte *pcPacket* array is defined in file `twd.c`, the ROM code size is 1518 bytes higher than if the same array is defined in the file `main.c`, together with all the other global data. This was despite the fact that the total object code size was the same in both cases, and seemed to be because the target file for the Turbo C linker was `main`. As such, all global data was defined in `main.c`.

5.3.3 Efficient Parameter and Local Variable Declarations

It is important to control the parameter and local variable declarations for two reasons. Firstly, they increase the space consumed on the stack, which requires a larger stack. Secondly, identifying how the stack will build up requires a study of the function call hierarchy, and thus it is difficult to safely estimate the required stack size.

The technique used to handle these problem was to avoid declaring any structures as parameters or local variables. If any such structures were required by routines, pointers to global variables were passed to / used by the routines. If the required global variables did not exist, the global variables were declared – the RAM size used would be the same, but calculating how much RAM space is required became simpler.

5.3.4 Merging of Similar Functions

While designing the new remote boot system, it was observed that many functions would do similar tasks with some minor variations. Software enginnering principles state that such functions should not be merged. However, since this remote boot application is special in that it requires code size optimization, such functions were merged in the remote boot client code. These include all the BOOTP and new `pcboot` communication routines and the disk sector read and write routines. A lot of code size was saved as a result.

5.3.5 Elimination of Pseudo-Redundant Code

'Pseudo-redundant' code is not standard terminology, but is introduced here to signify code that seems to have no redundancies, but actually has redundancies. Finding and eliminating pseudo-redundant code gives no major gain at any one place, but if done throughout the code, it gives significant advantages. However, such optimizations should not unduly increase the code complexity. A few examples are given below.

- When initializing complex data structures, most fields and arrays are initialized to null. Setting each field and each element of each array to null results in redundant code. The whole structure can be initialized to null in one loop, and the non-null fields can be filled later.

- When printing strings, such as for diagnostics, often different conditionals print strings which have matching substrings. As strings consume much more data space than the code space used by either conditionals or print function calls, a lot of space is saved by restructuring the conditionals to print a given substring only once.

- In certain types of code, such as ethernet card drivers, there are long chains of the same function call with different parameters. By using arrays to store the parameters of the function calls, and using a loop to execute the function calls with the different parameters, it was found that over half the originally used code and data space was saved (for chains with more than 10 consecutive instructions). This approach was picked up from `linux` source code, [Don92], [Don93], and was used in the ethernet card drivers for both the `wd8003e` and `ne2000` cards.

5.4 Transparency to PC Disk Configuration

There are some precautions that must be taken to ensure that the remote boot product is transparent to the PC disk configuration. These will ensure that all disks

present on the PC will still be available after a remote boot, and that the remote boot is possible even if the PC is diskless. These are described below.

5.4.1 New Disk Interrupt Handler

The new BIOS 0x13 disk interrupt handler needs to provide many different services. During the booting process, these are reset (0x00), read (0x02), write (0x03), get parameters (0x08), get changeline support (0x15) and get changeline status (0x16). These must provide the correct interface and return values when called. Complexity arises in achieving this for the get parameters service (0x08), which is described here.

The parameters of interest are the disk parameter table segment and offset, the maximum cylinder number, the maximum sector number, the number of heads and the number of drives.

The location of the disk parameter table is specific to each PC's BIOS configuration. The same location must be returned by the old and new BIOS 0x13 disk interrupt handlers, and so it is necessary to initially call the old BIOS 0x13 service 0x08 before replacing it with the new interrupt handler, and to store the data thus obtained. In this way the correct location of the disk parameter table can be returned.

The maximum cylinder number, the maximum sector number and the number of heads must be corresponding to those on the remote boot file, else the data offsets in the remote boot file cannot be correctly calculated. These parameters are obtained by initially reading the boot sector (logical sector 0) of the boot file, and extracting the required boot parameters.

The values listed in the above paragraph for the remote boot file and for the disk supported on the local A: drive may not be the same, e.g. a 1.44 MB boot file may be used to boot a PC with a 1.2 MB A: drive. This causes no problems on MSDOS, as MSDOS later re-reads the drive parameters and sets them to the local drive paramenters. The behavior of other operating systems in this regard is not known. This may result in the A: drive being inaccessible after booting operating systems other than MSDOS.

Finally, the issue of the number of drives to be returned is left. The original BIOS `0x13` disk interrupt handler will return 0 for a diskless PC, 1 for a PC with 1 or more floppy drives, and 1+`N` for a PC with `N` hard drives. For the remote boot application, a minimum value of 1 must be returned even for a diskless PC, else the A: drive required for remote boot will not be recognized. If the PC has 1 or more drives, the true number of drives must be returned, to make sure that the RAMDrive created is not given the same name as that of an existing hard drive. This issue is dealt with in more detail below.

5.4.2 Restoring Old Disk Interrupt Handler

The old disk interrupt handler must be restored at some stage of the remote boot, so that the original disk drive services supported by BIOS are available to the PC. This is done by using the new BIOS `0x13` disk interrupt handler non-standard service `0x1b`. This service restores the old disk interrupt vector in the interrupt vector table. The service `0x1b` can be used because BIOS does not define or reserve its usage. As the remote disk interrupt handler can no longer be accessed after a call to this service, the client code can be removed from the RAM, and the RAM space freed. However, this does not make any additional memory available to MSDOS, as MSDOS checks the maximum RAM available only once at the time of boot. The BIOS `0x13` service `0x1b` should be called when the A: drive is no longer required for booting. How this is done is discussed in the next section.

The program `reset.exe` was written to call this service, and its call appears in the batch file `proceed.bat` in the remote boot file image. This is shown in chapter 7.

5.4.3 RAMDrive Creation, Identification and Use

The reason for the creation of the RAMDrive is discussed in chapter 2. For MSDOS, the creation of the RAMDrive is achieved by using a command in the `config.sys` file. This forms a part of the boot file image, as shown in chapter 7.

The name of the RAMDrive is not explicitly specified, but is calculated from the number of drives returned by the new BIOS `0x13` disk interrupt handler service `0x08` and from the number of MSDOS partitions on the hard drives. For example, a PC with no hard drives will create a RAMDrive at C:, and a PC with one hard drive with 2 MSDOS partitions or 2 hard drives with 1 MSDOS partition each will create a RAMDrive at E:. To use the RAMDrive in the batch file commands in the remote boot file, it is necessary to identify the location of the RAMDrive. This is achieved by scanning drives V: to C: until the first existing drive is found – this will always be the RAMDrive. The search must be in reverse order as access to a disk may result in a disk read error. For example, if D: drive has a bad DOS format, and the RAMDrive is created at E:, scanning from C: onwards will result in a read error at D:. The sample batch commands to do this, and to later use the RAMDrive, appear in the batch file `autoexec.bat` in the remote boot file. These are shown in chapter 7.

Only after all the required A: drive files have been copied on to the RAMDrive can the original BIOS `0x13` disk interrupt handler be restored. The booting proceeds using the files in the RAMDrive.

It should be noted that in order to achieve operating system independence, equivalent commands for the creation and identification of the RAMDrive must appear in the remote boot file image for the operating system being booted.

5.5 Rewriting DOS Functions

The DOS library functions used were existing in the earlier implementation of remote boot. Care had to be taken to ensure that no other DOS library functions were used. These functions included `kbdhit`, `inputch`, `cgetsne`, `cprintf` and `Move`. Some other useful functions written were `ntohs`, `ntohl` and `pcGetClockValue`. The `htonl` and `htons` functions were not required as they were the same as of the `ntohs` and `ntohl` functions.

5.6 Operating System Independence

To achieve guaranteed operating system independence for the PC remote boot system, a knowledge of various operating system internals is necessary. Since this was difficult to obtain for all operating systems, a simpler approach was used – no MSDOS based dependencies were introduced into the client ROM code. The basis for this is the assumption that all PC operating systems must be built upon BIOS functions. The only BIOS function remote boot modifies is the BIOS 0x13 disk interrupt handler routine. This meant that to ensure operating system independence care had to be taken in the design of the new disk interrupt handler. It happens that the precautions required to be taken for this are included in the precautions to be taken to ensure transparency to the PC disk configuration, as was discussed earlier.

However, the need for a RAMDrive to be created, identified and used during remote boot introduces an additional complexity. Each operating system will have different commands which are to be used to create and identify a RAMDrive. It is left to the installer of a new operating system to identify these commands and appropriately configure the remote boot file.

In addition, the local A: drive of a PC booting an operating system other than MSDOS may become unavailable. This will happen if after remote boot, upon accessing the A: drive, the operating system does not re-read the local A: drive parameters, and if the local A: drive parameters are different from those of the remote boot file.

Beyond this, no precautions or guidelines were found to be appropriate or generic enough to ensure support for operating system independence during PC remote boot. Some situations could not be handled. For example, if the boot sector parameters of the remote boot file are not recognized by BIOS as standard, a warning is printed and users are allowed to continue booting at their own risk.

5.7 Ethernet Card Probes

As explained in chapter 3, the two most widely used ethernet adapter cards are the `wd8003e` and the `ne2000`. The remote boot product should support both these cards. The remote boot ROM code should be able to identify which card is installed on the PC and at which I/O base address it is installed. Therefore it is necessary to use ethernet card probes. The example probes for the ethernet adapter cards were studied from the `linux` source code, [Don92], [Don93].

Card probes were implemented for both the ethernet cards at I/O base addresses between `0x240` and `0x380`. The `wd8003e` card probe was simple - just a test for a checksum match on the first 8 bytes read from the base I/O address, plus an elimination check for soundcards which use the same checksum match. The `ne2000` card probe was more complex - first a check for a dp8390 chip [NSC86], followed by an initialization of the dp8390, followed by reading in the first 16 bytes from the I/O base address, distinguishing between the `ne2000` and `ne1000` cards, and finally a check that bytes 14 and 15 both equal 57.

If neither card is found, the ROM code reports this error and the PC boot stops. If either card is found, that card is initialized and used for future MAC layer communication.

5.8 ne2000 Card Device Driver

The device driver for the the `ne2000` ethernet adapter card was developed and integrated with the remote boot product. The Crynwr source code, [CHDTN88], [HHC88], and the dp8390 technical reference, [NSC86], were used to develop the device driver. A general purpose `ne2000` packet driver takes up 6 – 7 kB of code size. However, the lack of ROM code space meant that a 2 – 3 kB device driver had to be developed. This was achieved by eliminating all functionality that would not be required by the remote boot system, such as multicast, receiver overrun check, asynchronous sends, etc. A cyclic buffer scheme was used to store the packets for transmission and the packets received. The dp8390's two DMA channels were used to transfer data from the `ne2000` card's local memory to the network FIFO, and from

the PC's main memory to the ne2000 card's local memory. Packets were formatted according to the IEEE 802.3 ethernet standard.

5.9 Ensuring Password Security

The user's login name and password were copied into RAM at multiple locations - in the buffers used to read the strings, in the packets created and returned for authentication, in the packets created and returned by the UDP layer and in the packets created and returned by the MAC layer. As the RAM space used by the remote boot system is never reused, these strings would remain in certain locations at the top of the memory, and could be searched for and read by a PC user. To eliminate this security loophole, all copies of the packets used at various layers were cleared to null after a successful authentication.

Further, earlier these strings were sent unencrypted over the network, and a user sitting with a network snooping program such as tcpdump could scan the authentication packets and extract the login names and passwords. To reduce the chances of this happening, a simple but unusual encryption algorithm is used to encrypt the password at the client side and decrypt is at the server side. This makes the location and identification of the login name and password difficult for a malicious network snooper. However, decoding by a network snooper is still possible. The coding of algorithms required to make security foolproof would require substantial programming effort.

5.10 Improved Diagnostics and Error Handling

Diagnostics is an area where substantial coding effort has gone. About 15 % of the total ROM code size consists of error analysis, error reporting, and general status reporting. All the diagnostics cannot be described here, but a few examples are given to show how diagnostics were coded to improve usability and maintainability. In addition, the layout and line spacing of all messages has been designed to make them pleasant to read.

- Even before the BOOTP server is contacted, information relating to the ethernet adapter card is printed on the screen. Thus, if the PC has an ethernet card which is not supported by the remote boot system, or will be incompatible with the packet driver installed by the remote boot file, the relevant information is reported to the user right at the beginning, before the ethernet adapter card is used for booting. The ethernet address of the PC is also reported, to allow the PC's entry in the `bootptab` database to be made simply and quickly.

- After the BOOTP server is contacted, the internet address of the PC, its host name, and the name of the boot file that will be used for booting are printed on the screen. These help to identify the PC in the `bootptab` database, help to easily diagnose booting problems due to an incorrect boot file specification, and serve as a simple check that syntactically correct BOOTP information was received by the PC.

- A distinction is made between errors arising out of a server not being contactable, and a server reporting back a failed operation. This helps tracing of booting errors more easily.

- If the boot file does not correspond to a standard BIOS floppy type, this is reported to users, and they are advised to reboot. However, users also are give the option of continuing with the boot, even though it may fail.

- If an entry in `bootptab` has configured a PC to do a local disk boot, the user is asked to prompt the PC to continue. This is so that the initial ethernet card information and the BOOTP information can be viewed by the user, and so that the user knows that a local disk boot is about to take place.

There are many other places where similarly intelligent diagnostics and error handling have been built into the remote boot system.

5.11 Terminate and Stay Resident Client Code

The earlier remote boot implementation had provision for a TSR client executable. With the new implementation, the features which were to be tested by the TSR code had to be supported and extended.

The major new features which were added to the TSR code were the display of the BOOTP packet parameters, the display of the remote boot sector parameters, a step by step remote boot implementation which stops at key points in the client code, and a display of the packet header information in the ethernet, internet and UDP layers.

To save space in the ROM code, these features are available only with special compilation using the *TEST* and *DEBUG* flags. The features they support are not meant to be programmed into a ROM, and they are only used for testing and debugging the remote boot system as a user program.

Chapter 6

Software Quality Implementation Details

In modern software products, the cost of maintaining and upgrading a software product over its life-cycle is 3 to 5 times the cost of development of the software product. In turn, the cost of development of a software product is about 10 times higher than the cost of the development of a functional program [Jal91].

The starting point of this thesis work was a remote boot system, which had to be re-designed to overcome several limitations, which had to be made into a viable software product, and whose future usage and maintenance costs had to be minimized. From the facts given in the above paragraph, it is clear that a great deal of care and effort had to go into the software engineering aspect of this work. That is why a complete chapter has been devoted to this aspect of the work.

6.1 Guidelines for High Quality Code

In order to write high quality code, some standard guidelines have to be followed. These guidelines are based on my industrial experience at Inter Software and Technologies, Pvt. Ltd, and on the specifications in [Jal91], and have been strictly adhered to in the new remote boot implementation.

6.1.1 File and Function Headers

All files have headers to summarize their purpose, contents and history. An example file header is given here.

```
/*************************************************************************/
/*                                                                       *
 * Filename:          tinyudp.c                                          *
 *                                                                       *
 * Owned by:          IIT Kanpur                                         *
 *                                                                       *
 * Last mod. date:    27th May 1996                                      *
 *                                                                       *
 * Contents:            This file contains the routines implementing UDP *
 *                    and IP for the remote boot system. In addition, the *
 *                    UDP application routines for BOOTP, authentication *
 *                    and boot file service are implemented here. Several *
 *                    test and debugging routines are also in this file.  *
 *                                                                       */
/*************************************************************************/
```

Similarly, functions have headers which allow the reader to understand the interface and purpose of a function without going over the code. An example function header is give here.

```
/*************************************************************************/
/*                                                                       *
 * Function name:     udpInit                                            *
 *                                                                       *
 * Arguments:                                                            *
 *      arg1:         p_pcUdpSocket - s:  pointer to the socket structure *
 *      arg2:         pcIpHwAddress - inaddr:  IP address of destination  *
 *      arg3:         word - localPort:  local port number                *
```

```
*       arg4:          word - foreignPort:  foreign port number          *
*                                                                        *
* Return value:       void                                             *
*                                                                        *
* Purpose:              This function fills in the socket structure with *
*                     the necessary parameters. The s_pcUdpSocket        *
*                     structure is that of a UDP socket.                 *
*                                                                        */
/************************************************************************/
```

6.1.2 Header File Partitioning

Guidelines for header file partitioning include several things. Firstly, design of header files such that the information contained in them is cohesive. This in turn makes the dependencies on header files simple and easy to understand. Secondly, separate inclusion of the operating system header files and the local header files. Thirdly, avoiding hierarchical inclusions of local header files, as these make dependencies difficult to understand.

Some of the header files used and their inclusion is shown in an example here.

```
/********************** Standard Include Files ***************************/
#include <conio.h>
#include <dos.h>
#include <bios.h>
/********************** Standard Include Files ***************************/

/************************ Local Include Files ****************************/
#include <pcboot.h>
#include <handler.h>
#include <tinyudp.h>
/************************ Local Include Files ****************************/
```

6.1.3 Macro, Typedef, Function and Variable Declarations

Good guidelines for the declarations of macros, typedefs, functions and variables make the code easy to read and understand, and thus easy to maintain.

For macros, fully descriptive names (in capitals with separating underscores), are used to make macros stand out from variables.

```
/********************************* Macros **********************************/
#define  PC_TIMER_INTERRUPT          0x1a
/********************************* Macros **********************************/
```

For typedefs, fully descriptive names in small case, with an s_ or p_ prefix to distinguish structures from pointers are used. Capitalization serves the purpose of the underscores in macros. For the variables in typedef structures, the guidelines for local variables are followed.

```
/************************ Data Type Definitions ****************************/
typedef struct
{
    longword        authcode;
    byte            loginId [PC_MAX_LOGIN_ID_SIZE];
    byte            password [PC_MAX_PASSWORD_SIZE];

} s_pcAuthInfo, *p_pcAuthInfo;
/************************ Data Type Definitions ****************************/
```

For functions, a clear distinction between global, external and static functions is made. All functions are to be kept static if possible. Function and argument names are fully descriptive. Capitalization replaces the function of underscores in macros. Global functions start with a uniform prefix, which is pc for the PC remote boot system.

```
/******************** Global Function Declarations ************************/
void                pcStringCopy (char *dest, char *source);
```

```
/********************* Global Function Declarations ************************/

/******************** External Function Declarations ***********************/
extern longword    pcClockValueRough (void);
/******************** External Function Declarations ***********************/

/********************* Static Function Declarations ************************/
static int         bootpOrPcbootRoutine (pcIpHwAddress inAddr, word
                                         localPort, word foreignPort,
                                         p_pcInfo pcbootInfo, p_pcBootpInfo
                                         bootpInfo);
/********************* Static Function Declarations ************************/
```

For variables, the naming guidelines are similar to those for functions. Variables are distinguished as global, extern, static or local. The first two start with the same pc prefix. Names are descriptive and in small letters. Capitalization replaces the function of underscores in macros.

```
/********************* Global Variable Declarations ************************/
s_pcUdpSocket      pcUdpSocket;
/********************* Global Variable Declarations ************************/

/******************** External Variable Declarations ***********************/
extern byte        pcBootfileName [PC_MAX_FILENAME_SIZE];
/******************** External Variable Declarations ***********************/

/********************* Static Variable Declarations ************************/
static word        userAuthenticated;
/********************* Static Variable Declarations ************************/

int pcInitialise ()
{
    /*
```

```
    * Local variables
    */
   word        localPort;
   word        foreignPort;
   :
   :
}
```

6.1.4 Design Modularity: Coupling and Cohesion

Coupling is a measure of the interdependence between functions. Cohesion is a measure of how tightly bound the elements of a function are to one another. The goals of low coupling and high cohesion have been followed, except in cases where these requirements cause a conflict with the design limitations and specifications.

Thus, by and large the function design is modular, apart from aspects such as a single authentication and boot file service which was required to minimize the number of non-standard services, and merged functions which were required to reduce the ROM code size, etc.

6.1.5 Conditionals and Control Flow

A good design of conditionals and their resulting control flow makes code easy to read and follow. The goal is to make conditionals as simple as possible, define logical order of choices as naturally as possible, use only explicit comparisons, use brackets clearly to define arithmetic order, use parenthesis to mark conditional code, and restructure code to avoid nesting over 2 - 3 levels deep. The following badly structured code is rewritten to show the effect on readability of good conditionals and control flow.

Original code:

```
rxMitTime = pcClockValueRough() + PC_UDP_WAIT;
while (rxMitTime > pcClockValueRough())
    if (!(ipHeaderPtr = (p_pcIpHeader)neIsPacket()))
```

```
            if (checkPacket((byte *)ipHeaderPtr, 0x800))
                if (bootpOrPcbootHandler(ipHeaderPtr, pcbootOpcode,
                bootpOpcode, isPcboot))
                    return(1);
```

Rewritten code (without comments):

```
rxMitTime = pcClockValueRough() + PC_UDP_WAIT;
while (rxMitTime > pcClockValueRough())
{
    ipHeaderPtr = (p_pcIpHeader)neIsPacket();
    if (ipHeaderPtr == NULL)
    {
        continue;
    }
    if (checkPacket((byte *)ipHeaderPtr, 0x800) != 1)
    {
        continue;
    }
    if (bootpOrPcbootHandler(ipHeaderPtr, pcbootOpcode, bootpOpcode,
    isPcboot))
    {
        return(1);
    }
}
```

6.1.6 Spacing, Line Wrap and Comments

Though these seem trivial aspects to use guidelines for, these items significantly affect code readability, and thus the cost of maintenance.

For spacing, a uniform indentation of 4 white spaces is used for each additional scope or conditional level (and not at any other place), and the area is marked out

by parenthesis on separate lines. No tabs appear in the code. Space is left before conditionals, after commas, and on either side of arithmetic operators.

For line wrap, no line is longer that 80 characters, even if it means leaving large parts of the line blank or breaking up strings across lines. This ensures that a printout of the code on any printer will be fully readable.

Comments are the most important aspect for readability. They follow a uniform pattern leave a blank line and then to indent to the same depth as the code to be explained. Comments explain the action being taken together with the reasons. Long comments are used if some unusual behavior is to be explained.

```
/*
 * Call pcInitialise again to reinitialize parameters
 * This is required because when the PC is on and waiting for the
 * login, after 5 - 10 minutes data corruption occurs (maybe due to
 * the other interrupts meanwhile handled by the PC), due to which
 * the pcboot server cannot be contacted. Resetting the PC removes
 * the problem. Instead of that, recalling pcInitialise here will
 * have same effect
 */
while (pcInitialise() == 0);
```

6.2 Product Documentation

Code will be read by persons who are to maintain or upgrade a system. But the users of the system should never have to go through the code to understand it. There should be complete user documentation of all aspects of the system use, describing the hardware and software requirements, installation and usage procedures, and the supported features of the system.

The documentation written for the remote boot system is centralized in one place in the Appendix to this thesis. It is arranged in a logical sequence of sections and is structured as a step by step set of instructions and trouble shooting options.

6.2.1 Documentation Provided

The documentation in the Appendix has a section on the hardware and software requirements, on the diskettes provided with this thesis and on the installation of the server. It has a section on the installation of the client – the creation of the ROM code to download, its fusing into the ROM, and the ROM's installations in the PC. The Appendix also has a section on the boot file creation and installation. All sections are complete in themselves, that is, all changes that are related to a section are specified within the same section.

The Appendix is intended to serve as the complete `PC Remote Boot Installation Manual`, which should be supplied to system administrators wishing to install PC Remote Boot on their network of PCs.

Chapter 7

The Remote Boot File

The remote boot file is not just an image of any bootable floppy – it must perform special operations specifically for remote boot. Thus its creation is not straightforward, and requires to be explained. This is done in this chapter, so that the design and implementation details of the remote boot file are centralized in one place.

7.1 Definition of the Remote Boot File

The remote boot file is a binary dump of all the sectors of a bootable floppy (i.e. one which can be used to boot a PC). The dump is in the logical order of sectors, e.g. on a 1.44 MB floppy, track 0, head 0, sector 1 to track 0, head 0, sector 18, and so on to track 79, head 1, sector 18.

The boot floppy used to make a remote boot file must have, in addition to the normal booting operations, certain commands which are specifically required for a remote boot. Thus, for remote boot to be implemented on a network of PCs, the normal booting floppy must be taken and modified before a dump of the floppy can be used to create a remote boot file.

7.2 Restrictions on the Remote Boot File

There are a few restrictions on the remote boot file, as pointed out above. These restrictions are discussed here.

7.2.1 Error Free Floppy

The floppy used to create a remote boot file must have no bad sectors. This is not an inherent limitation in remote boot, but due to the fact that a remote boot file can be dumped back onto a floppy to inspect it. Take the case that the original floppy used to create the remote boot file had a bad sector at location X, that the new floppy used to inspect a reverse dump of the remote boot file has a bad sector at location Y, and that X is different from Y. Then the remote boot file will be incorrectly dumped back onto the second floppy, and therefore it cannot be correctly inspected. As such, it is necessary that all floppies used to create or inspect a remote boot file have no bad sectors, i.e. high quality floppies should be used for these purposes.

7.2.2 RAMDrive Creation, Identification and Use

This aspect of the restrictions on the remote boot file has been already discussed in chapter 2 and chapter 5. As such, it will not be discussed again here. Examples of the commands that are used on a MSDOS boot floppy to create, identify and use a RAMDrive are given later in this chapter. It should be noted that for each operating system for which remote boot is to be implemented, if a RAMDrive is required, its equivalent commands must be added to the boot floppy before creating the remote boot file.

7.2.3 Restoring Old Disk Interrupt Handler

This aspect of the restrictions on the remote boot file has also been discussed in chapter 2 and chapter 5. An example of the command that is used on a MSDOS boot floppy to restore the old BIOS `0x13` disk interrupt handler is given later in this chapter. It should be noted that the `reset.exe` program called to restore the old

handler remains the same across operating systems, and only needs a recompilation for the new operating system for which remote boot is to be implemented.

There is no restriction on the floppy type used to create the remote boot file. All floppies in standard use, i.e. 360 kB, 720 kB, 1.2 MB and 1.44 MB can be used to create the remote boot file, and the choice will not affect the availability of the PC's A: drive after booting.

Because files are stored on a floppy in order of the logical sectors, and because the binary dump of a floppy is also taken in the same order, the remote boot file can be much smaller than the floppy size. So if all the booting files of a 1.44 MB boot floppy fill in only the first half of the disk space, the remote boot file can be as small as 722 kB in size.

7.3 An Example Boot File

Since the correct creation of a remote boot file is a critical and slightly obscure step in the installation of the remote boot system, an example is used to show how the boot file should be structured. This example should be studied carefully before creating and installing a new remote boot file. The example is for MSDOS on the top of which XFS is to be run. For other operating systems logically equivalent restrictions (as discussed above) hold true.

7.3.1 The Root Directory

The root directory contains the standard system configuration file, `config.sys`, the default MSDOS shell `command.com`, the standard startup batch file `autoexec.bat`, a new batch file `proceed.bat`, the executable to restore the old BIOS `0x13` disk interrupt handler `reset.exe`, and the `dos`, `xfs` and `bin` directories. Two files, `io.sys` and `msdos.sys` are actually the first two files in the root directory, but are hidden files, as specified by their attributes.

7.3.2 config.sys

This is the standard system configuration file used by MSDOS at boot time. It is normally optional, but as remote boot requires it for the creation of the RAMDrive, it is an essential file in the remote boot system. The fully commented file is shown below.

```
rem ** Here A: refers to the remote boot image
DEVICE=A:\DOS\HIMEM.SYS /TESTMEM:OFF
DEVICE=A:\DOS\EMM386.EXE RAM X=B800-C7FF  X=D000-D7FF
rem ** Above two lines to be commented if there is no extended memory
DEVICE=A:\DOS\ANSI.SYS
SHELL=A:\COMMAND.COM A:\ /E:1024 /P
BUFFERS=20
FILES=40
DOS=HIGH,UMB
rem ** Above line to be commented if there is no extended memory
LASTDRIVE=V
STACKS=9,256

DEVICE=A:\DOS\RAMDRIVE.SYS 168 512 16 /E
rem ** Creates a RAMDrive at C:, D:, E:, ... depending on PC hard disk s
rem ** disk:: 0 MSDOS partitions => c:, 1 => d:, 2 => e:, ...
rem ** /E flag to be removed if there is no extended memory
```

7.3.3 command.com

This is the default MSDOS shell. It is a program running under the control of MSDOS which provides the user's interface to the operating system. It parses and carries out the user's commands, including the loading and execution of programs from a disk or other mass storage device. It can be replaced with a shell of the users's own choice.

7.3.4 autoexec.bat

This is the standard startup batch file called by MSDOS at boot time. It is also an essential file in the remote boot system, as it identifies the RAMDrive and duplicates those contents of A: drive on to the RAMDrive which are required for the continued booting of MSDOS and the installation of XFS. The fully commented file is shown below.

```
@echo off
ver
prompt $p$g

rem ** Identify RAMDrive 'dl' (last drive)
set dl=none
if "%dl%"=="none" if exist v:\nul set dl=v:\
if "%dl%"=="none" if exist u:\nul set dl=u:\
                          :
                          :
if "%dl%"=="none" if exist d:\nul set dl=d:\
if "%dl%"=="none" if exist c:\nul set dl=c:\
rem ** Identifies RAMDrive at V:, ..., D:, C: depending on PC hard disk
rem ** This allows the use of a single boot image for any hard disk set
rem ** Is complex because a bad disk before the RAMDrive will report er

copy command.com %dl% > nul
copy proceed.bat %dl% > nul
md %dl%xfs > nul
copy \xfs %dl%xfs > nul
rem ** Contents of remote boot floppy copied onto RAMDrive
rem ** Booting will be from RAMDrive

set COMSPEC=%dl%command.com
%dl%proceed
```

7.3.5 proceed.bat

This is a new batch file called at the end of autoexec.bat in the remote boot file. It is also an essential file in the remote boot system, as it restores the old BIOS 0x13 disk interrupt handler. After this call, XFS is set up, and then the network specific initializations are done. The fully commented file is shown below.

```
path=%dl%xfs
%dl%
rem ** Comes out of the remote A: drive and into the RAMDrive

a:\reset
rem ** Last line restores the PC's local disk interrupt handler
rem ** Only the PC's local A: drive can be accessed after this

wd8003e 0x60 0x2 0x280
rem ** ne2000 0x60 0x3 0x300
winpkt 0x60
xfskrnl 0x60
xfstool @%dl%xfs\init
winpkt 0x62
rem ** Last 6 lines set up wd8003e / ne2000 packet drivers and XFS

g:\rboot\autoexec
rem ** Last line specific to the network setup in CSE Deptt, IIT Kanpur
```

7.3.6 reset.exe

This executable in the root directory of the boot file is called to restore the old BIOS 0x13 disk interrupt handler, by calling the 0x1b service of the new BIOS 0x13 disk interrupt handler. This call must occur in proceed.bat, after the RAMDrive has copies of all files required for the remaining boot, and before remote boot's device driver for the ethernet card is replaced by the new permanent packet driver. The

second restriction is because after installing the permanent packet driver, `reset.exe` will not be accessible to restore the A: drive interrupt handler on the PC.

7.3.7 The dos Directory

This directory contains system and executable files required for the specific DOS configuration to be set up on the PC. The file `ramdrive.sys` is the standard DOS system file to setup a RAMDrive. The file `ansi.sys` it the standard DOS system file to support ANSI terminal emulation. The file `himem.exe` is the standard DOS executable to manage areas of extended memory. The file `emm386.exe` is the standard DOS executable to simulate expanded memory and provide access to the upper memory areas.

7.3.8 The xfs Directory

This directory contains system and executable files required for the XFS configuration to be set up on the PC, and for the installation of packet drivers. The file `init` is the standard XFS system file to store the network drive initialization information. The file `hosts` is the standard XFS system file to identify recognized hosts on the network by their internet addresses. The files `xfskrnl.exe` and `xfsinit.exe` are the standard kernel and initialization programs of XFS. The files `wd8003e.com` and `ne2000.com` are the standard `wd8003e` and `ne2000` ethernet adapter card device drivers, only one of which is installed by `proceed.bat`. The file `winpkt.com` is the standard windows virtual packet driver which must be installed over the `wd8003e` or `ne2000` packet driver.

7.3.9 The bin Directory

This directory is not used by remote boot. It is used as a store for executables that may be required when the boot floppy is used to boot a PC from its local drives. These executables can also be run during the TSR version of remote boot, in which the remote A: drive can be explicitly accessed by the user.

7.4 Utilities to Create Remote Boot Files

Two utilities, `scand2f` and `scanf2d`, were written to create a boot file from a disk, and to write back a boot file to a disk. The second utility is necessary as an existing boot file may need to be inspected. Both these utilities can handle any floppy type (360 kB, 720 kB, 1.2 MB and 1.44 MB) in any floppy drive. They have been designed to be easy and fast to use. Their usage is discussed in more detail in the Appendix.

Chapter 8

Conclusions

This chapter deals with the final remote boot system status, and the results of re–design in terms of improvement of certain parameters of performance as compared to the earlier implementation. In addition, some suggestions are made for possible future extensions of the remote boot system.

8.1 Product Status

The remote boot system developed is a fully reliable system to authenticate users and boot PCs, on a LAN having a UNIX server, from the network. It is transparent to the PC configuration in terms of hard disks and ethernet adapter cards (for the most commonly used wd8003e and ne2000 cards). The product has been in use on 5 PCs in the Department of Computer Science and Engineering, IIT Kanpur for over 4 months. For the last 3 months, there have been no complaints reported by the PC users or system administrators. As such, it can be considered a stable product.

The re-designed PC remote boot system has also been installed on over a dozen PCs in the Computer Center of IIT Kanpur, and no complaints have been reported for 3 weeks.

8.2 Modes of Operation

As discussed in the earlier chapters, there are two modes of operation of the client ROM code: (1) a Terminate and Stay Resident version of the program that can be run as a user program for testing of code changes in the client code, (2) a fully operational ROM version which is fused into a ROM and plugged into the network interface card of a PC to make it boot from a remote server.

8.3 Parameters of Performance

A large amount of new functionality has been added in the re-designed remote boot system as compared to the earlier implementation. It is useful to see how these changes have affected the performance of the system in terms of various key parameters.

- **Code, Data and Stack Size:** The earlier implementation reserved 18 kB of RAM for code, data and stack. The re-designed version, despite the additional functionality, also reserves only 18 kB of space in the RAM for code, data and stack. The sizes of some of the important client object files and the ROM download file are listed below for the two implementations.

FILE	EARLIER PC REMOTE BOOT	RE-DESIGNED PC REMOTE BOO
main.com	15.9 kB	13.2 kB
main.obj	3.3 kB	3.0 kB
tinyudp.obj	3.0 kB	5.3 kB
handler.obj	2.4 kB	2.3 kB
twd.obj	2.0 kB	1.8 kB
tne.obj	--	2.8 kB
tinytcp.obj	4.9 kB	--
arp.obj	1.0 kB	--

- **Booting Time:** The earlier implementation took 32–36 seconds to do a remote boot on a PC–386. This should be compared to the 52 seconds required to do a local boot from the floppy drive and to the 15 seconds to do a local boot from the hard drive. The re-designed system also takes about 30 seconds after authentication to do a remote boot. On a PC–486, remote boot takes 14 seconds compared to 42 seconds required for a floppy drive boot. The absence of performance degradation despite increased functionality is mainly due to the use of light weight boot file access protocols. The processing time taken at the server side of the remote boot application is about 1.5 seconds.

- **Security, Reliability and Robustness:** The system is almost fully secure, with a single loophole which is in the case of a dedicated network snooper picking up authentication packets, and then locating and decrypting the login names and passwords in them. This is an unlikely event but still possible. However, security is much improved over the earlier system, where login names and passwords could be read by anybody using the PC. The system has proved highly reliable, and its only point of failure is if the network is down. This is a significant improvement over the earlier implementation, which had multiple points of failure and was often unoperational due to server problems. If the network is down, the remote boot system cannot contact the BOOTP server and does not allow a local boot. In such a case, PC becomes unusable unless the boot ROM is physically removed from the PC. However, in such cases, as workstations also cannot boot and as network drives are inaccessible, the PCs' use would anyway be limited. As for robustness, the new remote boot system has no known bugs and implements a truly transparent remote boot, which was not the case earlier.

- **Management from Server Side:** The management of PC remote boot by system administrators from the server side is one of the major gains of the re-design of PC remote boot. As described in chapter 4, many aspects of the PC boot can be controlled from the standard BOOTP configuration file, `/etc/bootptab`. As the system administrators will already be using this file to control the booting of workstations, using it to control the booting of PCs

is only a marginal overhead. In the earlier system, this was a major overhead and often a bottleneck, due to the limited control provided.

- **Product Maintainability:** This has been one of the most important goals of the re-design of PC remote boot. In the earlier implementation, it took over 3 months to fully understand the code and installation procedures, and a few aspects remained unclear until very late in the thesis work. It is estimated that the fully documented code and detailed Appendix will cut this time down to a couple of weeks for future upgradation and maintenance, if required.

8.4 Restrictions

There is one restriction on the PC remote boot system which had not been anticipated. This relates to the ne2000 card support.

Using the ne2000 ethernet card driver, access to the BOOTP server and user authentication (using the authentication and boot file server) were successfully implemented. However, the loading of MSDOS from the remote boot file was not successful. During the booting sequence, MSDOS would hang at the instruction `OUT 85, AL`. Port 85 is a port for the DMA chip. No logical reason could be found for this behavior during remote boot with an ne2000 card, despite 6 weeks of study on this problem. The standard Crynwr packet driver, `ne2000.com`, gave the same problem. It should be noted that PC packet drivers are intended to be installed only after a PC has booted. It may be possible that there is an undocumented conflict which occurs if the ne2000 packet driver is installed before the PC has booted, as is required by the remote boot system. However, the source of such a conflict or the means of resolving it could not be found.

Therefore, it has been decided to restrict PCs having an ne2000 ethernet adapter card to the first two stages of remote boot, i.e. BOOTP server access and user authentication. The third stage of downloading the remote boot file is replaced by a forced local drive boot. This restriction on PCs having an ne2000 ethernet adapter card does not downgrade the utility of the PCs, as without using remote boot a local boot was anyway being done. Such PCs can however now access the BOOTP

server to configure their boot, and authenticate their users. Thus, the remote boot system can be installed on a heterogeneous network of PCs having both `wd8003e` and `ne2000` ethernet adapter cards.

8.5 Possible Future Extensions

A few possible extensions have been identified, on which future work can be done on the remote boot system.

- The `ne2000` card problem needs to be resolved. Two manuals, giving details of the `74LS612` chip (DMA's memory mapper), need to be referred to to diagnose the problem. These are:

  (1) *LSI Logic Data Book - 1986,* `[SDVD001]`,

  (2) *TTL Data Book Vol 2 (Std. TTL, S and LS) - 1985,* `[SDLD001]`

- A network snooper can, with sufficient dedication, break into the security of the system. If a secure encryption algorithm is used, such a possibility is eliminated. This however, may require up to a few kbytes of client code, and will probably be necessary if PC Remote Boot is to become a commercial product.

- Only 13.2 kB of the 16 kB ROM code space available has been used. The remainder can be used to implement additional functionality. It is possible that a standard protocol for user authentication becomes established in the future, even though none is defined today. In anticipation of this, the remote boot file access service can be redesigned to use the standard Network File System, as was discussed in chapter 5. It is estimated that less that 2 kB of client code would be required to implement NFS on top of RPC and XDR.

- On the client side, the routines in `twd.c` and `tne.c` contain probes to identify the ethernet card and its I/O base address. However, separate boot files must be used for different ethernet adapter cards at different I/O base addresses, so that the permanent packet driver is correctly installed. If the existing probes

are used in a program on the boot file to identify, locate and install the correct packet driver, a single boot file can be used for PCs with different ethernet adapter card configurations.

- As described in chapter 3 and chapter 5, true operating system independence requires a knowledge of the internals of a variety of operating systems. This is an area where more study should be done.

Appendix A

PC Remote Boot Installation Manual

This manual is meant for system administrators intending to install or maintain PC Remote Boot in the PCs on their network. It is also meant to be read by persons who need to maintain or upgrade the PC Remote Boot software, and thus need to be able to install and test any changes they make.

This manual contains a detailed step by step guide to the installation and maintenance of this product. It covers all the aspects of installation, in logically partitioned and individually complete sections. It includes trouble shooting measures in case problems are encountered in the installation.

Some paragraphs and sentences in this document are prefaced with the characters [*]. These deal with issues relating to the hardware available specifically in the Computer Science and Enginnering Department of the Indian Institute of Technology, Kanpur. Such an approach is necessary as PC Remote Boot has been developed using certain commercial hardware, and some essential information may be specific to the hardware used. It is expected (but not guaranteed) that this information will also be relevant when different but equivalent hardware is used.

In this manual there is some duplication of the material presented in the M.Tech thesis *A Reliable Network Boot Service for PCs*. This thesis, supervised by Dr. Rajat Moona, was submitted by Bhartendu Sinha at IIT Kanpur in July 1996. This

manual forms the *Appendix* to the above thesis. The duplication of material was necessary to make the manual an independent and complete document.

A.1 Hardware and Software Requirements

A.1.1 Hardware Requirements

- **Network:** A Local Area Network (LAN) on which the IEEE 802.3 ethernet protocol can be run.

- **Personal Computer(s):** One or more IBM compatible PCs with ethernet adapter cards connecting them to the LAN. The cards should be either `ne2000` or `wd8003e` ethernet adapter cards, or their compatibles. They must be able to support a boot ROM of at least 16 kB size. At least one PC must have a 1.44 MB 3.5 inch floppy disk drive, to read the `distribution diskettes`. At present, PCs having ne2000 cards are restricted to doing boot configuration and user authentication only. Therefore, PCs having ne2000 cards can configure their booting procedure and can authenticate the user, but they will boot only from their local disks instead of from a remote boot file.

- **UNIX Server:** A UNIX server machine connected to the LAN and supporting the TCP/IP protocol suite.

- **EPROMs of size 16 kB (27128A):** A supply of 27128A EPROMs. These have a 16 kB address space. Note: EEPROMs or EPROMs with a larger address space may also be used. As these are not dealt with in this manual, the use of 27128A EPROMs is strongly recommended.

- **EPROM Programmer:** An EPROM programmer which can program the 27128A EPROM. It must be interfacable with a computer from which it can download files, and it must be able to calculate its buffer checksum & modify its buffer data. [*] The *ESA UPAT* of Electro Systems Associates Pvt. Ltd. was used in IIT Kanpur's CSE lab.

- **EPROM Eraser:** An EPROM eraser which can erase data from the EPROM using ultraviolet radiation. This is necessary only if used boot ROMs are to be reprogrammed. [*] The EPROM eraser by Microtech Instruments and Controls was used in IIT Kanpur's CSE lab.

- **Error Free Floppies:** High quality floppies, i.e. having no bad sectors and a long life, should be available. These will be used to make copies of the `sample boot floppy`, and be used as boot floppies for creating and inspecting the boot file. PC Remote Boot can be installed by using only the two `distribution diskettes` supplied with this manual.

A.1.2 Software Requirements

- **Server Platform Software:** The UNIX server machine must support standard protocols recommended by the Internet Activities Board. In particular, the TCP/IP suite and BOOTP must be supported. If the server code is to be compiled, `make` utilities and a C compiler must be available.

- **Client Platform Software:** The PC must have standard ROM-BIOS support. If the ROM code or boot file creation utilities are to be compiled, `make` utilities, a `Turbo-C` compiler, assembler and linker, and the `exe2com` or `exe2bin` program must be available.

- **EPROM Programming Software:** The software to configure and control the EPROM programmer must be available. The EPROM programming software, the EPROM programmer and its adapter card would normally be supplied together.

- **Network Interface Card Setup Software:** The software to configure the `ne2000` or `wd8003e` cards should be present, unless these cards are configurable by manually operated DIP switches.

A.2 Distribution Diskettes

Two 1.44 MB 3.5 inch distribution diskettes are supplied with this manual. These are described in this section.

A.2.1 Client and Server Software Floppy

This floppy contains the source code for the PC Remote Boot client and server, and the binaries for the PC Remote Boot client. Server binaries are not supplied as they will be platform dependent. It also contains an example /etc/bootptab entries file, an example /etc/inetd.conf entries file, an example /etc/services entries file, and a copy of this manual in lj (laser jet) format. The directory structure with comments is given below.

```
manual.lj                   /* This manual in lj (laser jet) format */
bootptab                    /* Example /etc/bootptab entries */
inetd.con                   /* Example /etc/inetd.conf entries */
services                    /* Example /etc/services entries */

source                      /* Source code */
  -> client                 /* Client source code */
       -> main.c            /* ROM code main program */
       -> tinyudp.c         /* ROM code UDP routines */
       -> handler.c         /* New BIOS 0x13 interrupt handler routine */
       -> tne.c             /* ne2000 card driver routines */
       -> twd.c             /* wd8003e card driver routines */
       -> tests.c           /* Terminate and Stay Resident (TSR) routines
       -> pcboot.h          /* PC Remote Boot header file */
       -> tinyudp.h         /* UDP header file */
       -> handler.h         /* BIOS 0x13 interrupt handler header file */
       -> tne.h             /* ne2000 card driver header file */
       -> twd.h             /* wd8003e card driver header file */
       -> act.asm           /* ROM code routine for relocation */
```

```
        ->  newintr.asm     /* New BIOS 0x13 interrupt interface routine */
        ->  ourproc.asm     /* Some rewritten DOS functions */
        ->  c0.asm          /* Turbo C start up code */
        ->  rules.asi       /* Turbo C assembler rules and structures */
        ->  scand2f.c       /* Disk to boot file dump utility */
        ->  scanf2d.c       /* Boot file to disk dump utility */
        ->  reset.c         /* Old BIOS 0x13 interrupt restore utility */
        ->  make.rom        /* ROM code and utils creation makefile */
        ->  make.tsr        /* TSR code creation makefile */
        ->  make.dbx        /* Debug code creation makefile */

  ->  server                /* Auth. & boot file server source code */
        ->  pcboot.c        /* Auth. & boot file server main program */
        ->  pcboot.h        /* Auth. & boot file server header file */
        ->  makefile        /* Auth. & boot file server creation makefile *

binary                      /* Binaries */
  ->  client                /* Client binaries */
        ->  main.com        /* Downloadable ROM code executable */
        ->  act.bin         /* ROM code executable for relocation */
        ->  reset.exe       /* Old BIOS 0x13 interrupt restore utility */
        ->  scand2f.exe     /* Disk to boot file dump utility */
        ->  scanf2d.exe     /* Boot file to disk dump utility */
```

A.2.2 Sample Boot Floppy

This floppy is a sample MSDOS boot floppy, from which an MSDOS boot file can be created for use by PC Remote Boot. The directory structure, with comments is given below.

```
io.sys          (hidden file)  /* Resident device drivers */
msdos.sys       (hidden file)  /* MSDOS kernel */
config.sys                     /* System configuration file */
```

```
command.com                    /* Default MSDOS shell */
autoexec.bat                   /* System startup batch file */
proceed.bat                    /* Additional batch file */
reset.exe                      /* Old BIOS 0x13 interrupt restore utility *

dos                            /* MSDOS utilities directory */
    -> ramdrive.sys            /* RAMDrive creation file */
    -> ansi.sys                /* ANSI terminal emulation file */
    -> himem.sys               /* Extended memory management file */
    -> emm386.sys              /* Expanded memory simulation file */

xfs                            /* XFS directory */
    -> init                    /* Network initialization information */
    -> hosts                   /* Internet addresses of hosts */
    -> xfskrnl.exe             /* XFS kernel */
    -> xfsinit.exe             /* XFS initialization */
    -> wd8003e.com             /* wd8003e packet driver */
    -> ne2000.com              /* ne2000 packet driver */
    -> winpkt.com              /* Windows virtual packet driver */

bin                            /* DOS utilities directory (not accessed) */
```

A.3 Server Installation

This section deals with the setting up of the PC Remote Boot servers – the BOOTP server and the authentication & boot file access server.

A.3.1 Preparation of Server Binaries

Only the authentication & boot file access server is to be compiled, as the BOOTP server is already available on UNIX platforms. The procedure to prepare the authentication and boot file server binary is as follows.

- Boot a PC having a 1.44 MB 3.5 inch floppy disk drive. The `sample boot floppy` can be used to boot this PC.

- Copy the contents of directory `source\server` from the `client and server software floppy` to a directory which has been mounted on the UNIX server.

- On the UNIX server, enter the above mentioned directory. Remove all the occurrences of *CTRL-M*, in all the files present.

- Give the command *make clean install*. If errors occur at this stage, contact the supplier of the `distribution diskettes`. Otherwise the `pcbootd` binary is now installed as `/var/adm/pcbootd`.

  Note: `pcbootd` can be run as an `inetd` daemon process or as a normal user process. To run it as a normal user process, either the user must be root, or the program must be recompiled after changing the *PC_SERVER_PORT* in the server source code file `pcboot.h` to a value above 5000. To run it as an `inetd` daemon process, a `-i` flag must be added.

A.3.2 Setting up Server Configuration Files

There are three standard UNIX system configuration files that need to be modified to set up PC Remote Boot at the server side: `/etc/bootptab`, `/etc/inetd.conf` and `/etc/services`. The steps to modify these files are given as follows.

- Boot a PC having a 1.44 MB 3.5 inch floppy disk drive. The `sample boot floppy` can be used to boot this PC.

- Copy the `bootptab`, `inetd.con` and `services` files of the root directory from the `client and server software floppy` to a temporary directory which has been mounted on the UNIX server.

- Study these files and make the corresponding additions in the UNIX configuration files `/etc/bootptab`, `/etc/inetd.conf` and `/etc/services`. The `/etc/bootptab` file should be updated after reading the man page of `bootp`, and the additions made will be dependent on the local network environment.

The entries for the individual PCs will be described in sections A.4.5 and A.5.4. The latter two files, `/etc/inetd.conf` and `/etc/services` have standard additions. Examples for changes in all three files are shown here.

```
/etc/bootptab entries for remote boot:
#   Optional vendor specific flags for PC boot are explained below
#   T128=C0:\
#   b7: 0 => Local boot,          1 => Remote A: drive boot          DEFAULT:
#   b6: 0 => No authentication,   1 => Authentication required       DEFAULT:
#   b5: 0 => Remote A: drive      1 => Remote A: drive writable      DEFAULT:
#            never writable            before reset of disk handler
#   b4: 0 => No debug info        1 => Print debug info with Term.  DEFAULT:
#                                      & Stay Resident version
#   b3-b0 => Not used                                                DEFAULT:
#   T129=0092:\
#   Authentication + boot file server port number                    DEFAULT 0x0(
#   T130=9010A221:\
#   Authentication + boot file server IP address   DEFAULT  subnet broadc
#   NOTE: After getting the first valid reply,
#         the correct IP addr is set

.PC_DEFAULTS:\
        ht=ethernet:hn:sm=255.255.0.0:vm=rfc1048:\
#       ht: Ethernet LAN, having 8 byte hardware addresses
#       hn: Write host name in the BOOTP reply
#       sm: Subnet mask is 255.255.0.0
#       vm: BOOTP version corresponding to rfc1048
#       NOTE: hosts internet address is picked up from the NIS

#       T129=0092:\
#       T130=9010A221:\
```

```
/etc/inetd.conf entries for remote boot:
# Start the BOOTP server with debug level 2
bootps  dgram  udp  wait  root  /usr/sbin/bootpd  bootpd -d 2  /etc/bootp
# Start the authentication & boot file access server as an inetd daemon
pcbootd  dgram  udp  wait  root  /var/adm/pcbootd  pcbootd -i

/etc/services entries for remote boot:
# Define port numbers of the BOOTP server and client
bootps          67/udp
bootpc          68/udp
# Define port number of the authentication and boot file access server
pcbootd        146/udp
```

- After the above changes have been made, the `inetd` network services daemon must be restarted by *root*, by giving the command *kill -HUP <inetd_process_id>*. All the changes in the server configuration files are now effective, and the servers required for PC Remote Boot are installed.

A.4 Client Installation

This section deals with all aspects of the testing and installation of PC Remote Boot clients, i.e. the PCs.

A.4.1 Client Testing without a Boot ROM

The client code can be tested as a Terminate and Stay Resident (TSR) user program. If any changes have been made to the client code, they should be tested in this mode before fusing the client code into a boot ROM. A description of how to do this is given below.

- Boot a PC having a 1.44 MB 3.5 inch floppy disk drive. The `sample boot floppy` can be used to boot this PC.

- Copy the contents of directory `source\client` from the `client and server software floppy` to another directory on the above PC.

- Enter this above mentioned directory, and give the command *make -fmake.tsr clean all.* If errors occur at this stage, check if they are a result of any changes made in the client code or in the makefile. If they are not due to such changes, contact the supplier of the `distribution diskettes`. If no errors occur, the Terminate and Stay Resident binary `main.com` is now ready.

- Update the UNIX server file `/etc/bootptab` as described in sections A.4.5 and A.5.4.

- If changes were made to files existing on the network drives of the PC which was used to create the TSR binary, these network drives must be unmounted at this stage. This is because the running of the TSR program will reset the ethernet adapter card, and this will result in a loss of network cache coherence, due to which modified files will get severely corrupted. After unmounting, remount the network drive containing the TSR binary `main.com`.

- The client TSR binary is now ready to be tested. Give the command *main*, and observe the behavior of the client program.

  Note1: The run of the TSR binary leaves the user in the remote A: drive. The utilities in the `bin` directory can be run by the user. Write permission to the remote boot file is controlled by bit b4 in the *T128* option of the `/etc/bootptab` file, as shown earlier.

  Note2: In order to force a complete boot, the line *geninterrupt(0x19);* in the file `main.c` of the client code must be uncommented, and the TSR binary recompiled. Then the TSR program will take the PC through the complete BIOS bootstrap sequence.

  Note3: The `make.dbx` makefile is rarely used for compilation, as it results in the printing of a large amount of debugging information. However, the user may also wish to see these messages using the *-DDEBUG* option used in the `make.dbx` makefile.

A.4.2 Preparation of ROM binaries

The modified and tested client ROM code can be compiled to prepare binaries for downloading to the EPROM. These binaries, i.e. `main.com` and `act.bin` can also be picked up directly from the directory `binary\client` in the **`client and server software floppy`**. A description of how to compile these binaries is given below.

- Boot a PC having a 1.44 MB 3.5 inch floppy disk drive. The **`sample boot floppy`** can be used to boot this PC.
- Copy the contents of directory `source\client` from the **`client and server software floppy`** to another directory on the above PC.
- Enter this above mentioned directory, and give the command *make -fmake.rom clean all.* If errors occur at this stage, check if they are a result of any changes made in the client code or in the makefile. If they are not due to such changes, contact the supplier of the `distribution diskettes`. If no errors occur, the EPROM downloadable binaries, `main.com` and `act.bin`, are now ready.

A.4.3 Fusing a Boot ROM

The fusing of the downloadable client binary into an EPROM to create a boot ROM is a complex procedure, and is described below.

- On the computer which is to be used to run the EPROM programmer, mount the directory containing the `main.com` and `act.bin` downloadable binaries, or copy these binaries to a local disk.
- Set up the EPROM programmer. This is done by first attaching the EPROM programmer's adapter card to a computer and copying the EPROM programmer's software to the computer. The EPROM programmer should then be started up using the appropriate command. It may report a number of possible problems. The EPROM programmer's manual should be referred to to correct these problems.

[*] In IIT Kanpur's CSE lab, the major problem encountered initially was a conflicting device I/O base address. DIP switches on the adapter card and a **setup** utility available with the EPROM programmer software were used together to change this I/O base address. The details are supplied in the EPROM programmer manual.

- Once the EPROM programmer is in operation, check for its error free running. This is done by leaving the EPROM socket on the programmer empty and doing a *blank check* on the socket. If errors are reported, refer to the manual.

  [*] In IIT Kanpur's CSE lab, erroneous *blank checks* were handled by either tightening the cable between the EPROM programmer and its adapter card, or by leaving the programmer powered on and switching the computer off and then on again. The second approach was very reliable.

- Once the EPROM programmer is running without error, set it up to handle the 27128A EPROM. This is done by setting the *type* option (or equivalent option) to *27128A*. This is necessary to define the address range, voltage supplies and the programming algorithm. If the *27128A* option is not available on your EPROM programmer, PC Remote Boot 27128A ROMs cannot be fused.

- Now the EPROM programmer is set up to fuse 27128A EPROMs. A 27128A EPROM should be inserted into the socket on the EPROM programmer. The details to do this should be studied from the EPROM programmer manual.

  [*] In IIT Kanpur's CSE lab, this was done by aligning the EPROM's pins with the pin numbers marked beside the socket, and then pushing down a lever to lock the EPROM into the socket.

- Perform a *blank check* on the 27128A EPROM present in the socket. If an error is reported, the EPROM is not blank. It must be then left in the EPROM eraser for a specified amount of time to erase the data in it. After this, a *blank check* should report no error. If an error is still encountered, the EPROM programmer manual should be studied for possible errors in its use.

[*] In IIT Kanpur's CSE lab, the EPROM eraser was powered on with a setting of 70. The 27128A EPROM was then left in it face up for 45 minutes or more to erase all the data.

- After the *blank check*, remove the 27128A EPROM from the socket (this is important). The downloadable binaries can now be loaded into the EPROM programmer's *buffer*, using the appropriate option in the EPROM programmer software. Binary `act.bin` is to be downloaded at location 0, and binary `main.com` is to be downloaded at location 100 (hex).

- Obtain the value of the checksum of data in EPROM programmer buffer. If the EPROM programmer cannot display the checksum, it cannot be used to install PC Remote Boot. The checksum will be displayed either by a direct *checksum* option or by options such as *read* or *program*.

- If the checksum's last two digits are not 00, data in the EPROM programmer buffer must be modified to make the last two digits 00. This is done by displaying and then modifying byte number f0 (hex) in the buffer as per the following hexadecimal formula:

```
newValue = (oldValue + (0x100 - (checksum & 0xff))) & 0xff
```

  The checksum will now have the last two digits as 00.

- Now put the 27128A EPROM back in the EPROM programmer socket, and lock it in place. Then enter the option to *program* all the locations of the EPROM. No error should be reported. If an error occurs, study the EPROM programmer manual to check for incorrect usage.

- Remove the programmed 27128A EPROM from the socket. Now any number of 27128A EPROMs can be programmed using the same buffer contents by repeating the *blank check* and the last step.

A.4.4 Installing a Boot ROM in a PC

The programmed EPROM must be fitted into the boot ROM socket in the PC's network interface card (NIC), also known as the ethernet adapter card. The procedure below should be carefully followed, as incorrect steps can harm the user and damage the PC.

- Switch off the power supply to the PC and its monitor. Open and remove the cover of the PC, by pulling the cover back and then lifting it up. Touch the power supply box to remove any static charge.

- Disconnect the ethernet cable from the NIC, by gently turning its connector and then pulling it back. Pull the NIC from the I/O slot on the PC motherboard, and take it out.

- Put the programmed EPROM into the boot ROM socket in the NIC. The notch on the EPROM and on the socket must be on the same side. Care must be taken that all the pins make contact and that none of the pins get bent.

- Reverse the procedure described above to connect the NIC back to the ethernet cable and close the PC.

- Switch on power supply to the PC and the monitor. If the message "PC Remote Boot Version 2.0" appears on the screen, proceed to the next step. Otherwise remove the boot ROM from the NIC by reversing the above procedure. Do a local boot on the PC, and then use the NIC setup software or DIP switches on the NIC to configure the card as follows: boot ROM – enabled; boot ROM base address – `0xD000` (any non-conflicting setting will work); IRQ number – `0x2` or `0x9` (for `wd8003e`) and `0x3` (for `ne2000`); I/O base address – any location between `0x240` and `0x380`. If using the NIC `setup` software, save these changes to the NIC card, and test using `setup`. Re-install the programmed EPROM in the PC as described earlier. Upon switching on power to the PC and monitor, if the message "PC Remote Boot Version 2.0" still does not appear, there is likely to be a problem with either the NIC or the PC.

- If the message "Error: wd8003e and ne2000 cards not found" appears on your screen, the installed NIC is not supported by this version of PC Remote Boot. To use PC Remote Boot, a `wd8003e` or `ne2000` card or one of their compatibles must be used.

- The name of the card, its I/O base address and its ethernet address are reported on the screen. Note these down. This ends the sequence of operations that have to be done at the client side to install PC Remote Boot.

A.4.5 Updating Server Configuration Files

For each PC on which PC Remote Boot is to be installed, an entry has to be added to the UNIX server configuration file `/etc/bootptab`. These entries must be in addition to the *.PC_DEFAULTS* information added while setting up the server configuration files as described earlier in section A.3.2. Some example additions to this file are shown below.

```
# PCs with wd8003e cards must install the wd8003e packet driver
pc10:   tc=.PC_DEFAULTS:ha=00803c570040:T128=C0:bf="/usr/adm/xfswd.img":
pc37:   tc=.PC_DEFAULTS:ha=00803c57003a:T128=C0:bf="/usr/adm/xfswd.img":

# PCs with ne2000 cards must install the ne2000 packet driver
pc12:   tc=.PC_DEFAULTS:ha=00803c57002f:T128=C0:bf="/usr/adm/xfsne.img":
pc52:   tc=.PC_DEFAULTS:ha=0000E8C3A59b:T128=C0:bf="/usr/adm/xfsne.img":
```

A PC host name must be associated with its ethernet address. This address is reported by PC Remote Boot on the PC's screen. The boot file to be used for booting depends on the NIC installed, its I/O base address and its interrupt number, as was described earlier. This is further discussed in section A.5.

This ends the description of the PC Remote Boot client installation.

A.5 Boot File Installation

The server and client are both installed, but until the boot files specified in the `/etc/bootptab` file are installed, PCs will not boot. The `sample boot floppy` is a 1.44 MB 3.5 inch floppy which can be used to create boot file(s) required by PC Remote Boot. A copy of the original floppy supplied should be made, and the copy should be used for the operations described here. The creation and installation of the boot file is described in this section.

A.5.1 Modifying the Boot Floppy for the Local Network

The `sample boot floppy` contains files to do remote boot of MSDOS and install XFS on top of MSDOS. None of files on this floppy, except `proceed.bat`, `xfs\init` and `xfs\hosts`, need to be modified by the user. These 3 files need to be modified to set up the local network environment.

If some PCs on the network do not provide extended memory, changes also need to be made in the file `config.sys`. Because these changes are simple and well documented within the `config.sys` file itself, they are not explained here. Note: The RAMDrive on PCs with no extended memory is created in conventional memory. The remaining available memory is then insufficient for certain programs such as `telnet` and `ftp`. Therefore, such PCs should have extended memory installed if possible.

[*] In IIT Kanpur's CSE lab, XFS support is available using the PCNFS servers. If XFS is not to be used, modifications also need to be done on the file `autoexec.bat`, and the directory `xfs` may need to be replaced. Such changes should be done by a PC software configuration expert, and are not dealt with in this manual.

The file `proceed.bat` is given below.

```
path=%dl%xfs
%dl%
rem ** Comes out of the remote A: drive and into the RAMDrive


a:\reset
```

```
rem ** Last line restores the PC's local disk interrupt handler
rem ** Only the PC's local A: drive can be accessed after this

wd8003e 0x60 0x2 0x280
rem ** ne2000 0x60 0x3 0x300
winpkt 0x60
xfskrnl 0x60
xfstool 0%dl%xfs\init
winpkt 0x62
rem ** Last 6 lines set up wd8003e / ne2000 packet drivers and XFS

g:\rboot\autoexec
rem ** Last line specific to the network setup in CSE Deptt, IIT Kanpu
```

The packet driver installed, its interrupt number and its I/O base address must match with the configuration of the network interface card on the PC. This configuration was described earlier in section A.4.4. A different boot file must be created for each different configuration used. Because of this, it is advised that all NICs used on the PCs in the network be configured in the same manner. This will ensure that only two boot files – one for the `wd8003e` card and one for the `ne2000` card – will need to be installed.

[*] The last line of `proceed.bat` is specific to the network drive setup in IIT Kanpur's CSE lab. This will need appropriate modification on a different network. The `G:` drive was created by `xfsinit`, as specified in file `xfs\init`.

The file `xfs\init` is given below.

```
init BOOTP csum=off
pcnfsd cd2
login
mount    g: csp1:/g rsize=1024 wsize=1024
mount    s: cs21:/s rsize=1024 wsize=1024
mount lpt1: cs1:lp timeo=30
show
```

To take advantage of PC Remote Boot, there should be a common network drive available to all PCs. This common network drive will contain shared software, such as for `telnet`, `ftp`, `Turbo-C`, etc. This will allow a copy of commonly used utilities to be kept and maintained in a single place. Another common network drive can be used as a user writable area. Setting up such drives is advisable.

[*] The above file is used in IIT Kanpur's CSE lab to initialize XFS, identify the PCNFS server, prompt the user for a login and password, set up the `G:` and `S:` network drives, set up the printer `lpt1`, and show the XFS settings. These will need appropriate modification on a different network setup.

The file `xfs\hosts` is given below.

```
144.16.162.33       cd1
144.16.162.34       cd2
144.16.162.21       cs1
144.16.162.41       cs21
144.16.162.101      csp1
144.16.162.21       csesun1
144.16.162.101      csesparc1
144.16.162.254      cse_router_in
144.16.163.1        prithvi
144.16.163.4        vayu
144.16.163.20       ws12
144.16.160.228      ftp.iitk.ernet.in ftp.ee ee
```

[*] The above file is used in IIT Kanpur's CSE lab to identify the internet addresses of hosts which can be contacted by XFS. These entries must be modified on a different network.

A.5.2 Preparation of Boot File Utilities

There are certain utilities which are to be used to create or download the boot file. The source code of these utilities should not be modified by the user. These utilities, i.e. `scand2f.exe`, `scanf2d.exe` and `reset.exe` can be picked up directly

from the directory `binary\client` in the `client and server software floppy`. The creation of these utilities is also described here.

- Boot a PC having a 1.44 MB 3.5 inch floppy disk drive. The `sample boot floppy` can be used to boot this PC.
- Copy the contents of directory `source\client` from the `client and server software floppy` to another directory on the above PC.
- Enter this above mentioned directory, and give the command *make -fmake.rom clean utils*. If errors occur at this stage, check if they are a result of any changes made in the makefile. If they are not due to such changes, contact the supplier of the `distribution diskettes`. If no errors occur, the `scand2f.exe`, `scanf2d.exe` and `reset.exe` utilities are now ready.

A.5.3 Using Boot File Creation Utilities

Now that the utilities are available, they are to be used as follows.

- **scand2f.exe:** This is used to create a boot file from a boot floppy. The boot floppy should have been modified as described earlier to support remote boot. This utility can read 360 kB, 720 kB, 1.2 MB or 1.44 MB floppies from A: drive or B: drive and dump their image to a specified file. It gives diagnostics showing floppy status, copy status, and errors if any. The usage is simple and can be seen by giving the command *scand2f*. Boot files should be placed in a well defined location, such as `/usr/adm`.
- **scanf2d.exe:** This is used to create a boot floppy back from a boot file. It is used to examine the contents of an existing boot file. This utility can write to 360 kB, 720 kB, 1.2 MB or 1.44 MB floppies from a boot file, but the boot file's boot sector parameters must be the same as those of the floppy. This utility gives diagnostics showing boot file status, floppy status, copy status, and errors if any. The usage is simple and can be seen by giving the command *scanf2d*.

- **reset.exe:** This is used to restore access to the PC's local A: drive after booting using the remote A: drive. It must be called from file `proceed.bat` before any packet drivers are installed. No accesses can be made to the A: drive during the remaining booting process after `reset.exe` has been called. This file is available in the `sample boot floppy`, and is also supplied with the the `client and server software floppy`.

It should be noted that error free floppies should be used. If floppies having bad sectors are used to create or download boot files, unexpected behavior is possible.

A.5.4 Updating Server Configuration Files

The `/etc/bootptab` file should be updated if it is required that a PC boot from a different boot file. The PC identity, in terms of its ethernet address, internet address or hostname can be found by observing the initial PC Remote Boot diagnostics on the PC's screen. This is used to identify the PC's entry in the `/etc/bootptab` file. Then the string following the *bf:* option should be modified to the full pathname of the boot file to be used for booting. This pathname should be visible by the machine providing the authentication and boot file service.

The PC will now boot from the new boot file.

Glossary

ARP - Address Resolution Protocol

BIOS - Basic Input Output System

BOOTP - Boot Protocol

CSE - Computer Science and Engineering

CTRL - Control Key

DIP - Dual Inline Package

DMA - Direct Memory Access

DOS - Disk Operating System

EEPROM - Electrically Erasable Programmable Read Only Memory

EPROM - Erasable Programmable Read Only Memory

FIFO - First In First Out

FTP - File Transfer Protocol

HP-UX - Hewlett Packard UNIX

I/O - Input / Output

IAB - Internet Activities Board

IBM - International Business Machines

IEEE - Institution of Electrical and Electronic Engineers

IIT - Indian Institute of Technology

IP - Internet Protocol

IRQ - Interrupt Request

LAN - Local Area Network

MAC - Medium Access Control

NFS - Network File System

NIC - Network Interface Card

NIS - Network Information Services

PC - Personal Computer

RAM - Random Access Memory

RARP - Reverse Address Resolution Protocol

RFC - Request For Comments

ROM - Read Only Memory

RPC - Remote Procedure Call

SNA - System Network Architecture

SunOS - Sun Operating System

TCP - Transmission Control Protocol

TFTP - Trivial File Transfer Protocol

TSR - Terminate and Stay Resident

UDP - User Datagram Protocol

XDR - eXternal Data Representation

XFS - X File System

References

[Man94] T. J. Manjunath *Remote Booting of Networked PCs, M. Tech Thesis.* Department of Computer Science and Engineering, IIT Kanpur, February 1994

[Rav92] L. Ravichandar *Booting Diskless PCs from a remote UNIX server, M. Tech Thesis.* Department of Electrical Engineering, IIT Kanpur, June 1992.

[CG85] B. Croft and J. Gilmore *BOOTSTRAP Protocol (BOOTP), RFC 951.* Stanford and SUN Microsystems, September 1985.

[Prin88] P. Prindeville *BOOTP Vendor Information Extensions, RFC 1048.* McGill University, February 1988.

[Rey88] J. K. Reynolds *BOOTP Vendor Information Extensions, RFC 1084.* Information Sciences Institute, December 1988.

[Wim93] W. Wimer *Clarifications and Extensions for the Bootstrap Protocol, RFC 1532.* Carnegie Mellon University, October 1993.

[AD93] S. Alexander and R. Droms *DHCP Options and BOOTP Vendor Extensions, RFC 1533.* Lachman Technology, Inc. and Bucknell University, October 1993.

[StJ85] Mike StJohns *Authentication Server, RFC 931.* NIC, January 1985.

[SC81] K. R. Sollins, Noel Chiappa *The TFTP Protocol, RFC 783.* NIC, June 1984.

[Stev92] Richard Stevens *UNIX Network Programming.* Prentice-Hall of India Private Limited, 1992.

[Jal91] Pankaj Jalote *An Integrated Approach to Software Engineering.* Springer-Verlag New York Inc.

[Dun88] Ray Duncan *Advanced MSDOS Programming.* Microsoft Press, 1988.

[IBM84] *PC/AT Technical Reference.* International Business Machines Corporation, 1984.

[Nor91] Peter Norton *Inside the IBM PC.* Prentice-Hall of India Private Limited, 1991.

[Gil94] Frank V. Gilluwe *The Undocumented PC.* Addison-Wesley, 1994

[Stev89] Al Stevens *TURBO C: Memory Resident Utilities, Screen I/O and Programming Techniques.* Tech Publications, 1989.

[NSC86] *Series 32000 Databook, 1986.* National Semiconductor Corporation.

[CHDTN88] Bob Clements, Eric Henderson, Dave Horne, Glenn Talbott and Russell Nelson *dp8390 assembly language device driver: Source Code.* Crynwr Software Inc.

[HHC88] David Horne, Eric Henderson, and Bob Clements *NE2000 assembly language routines: Source Code.* Crynwr Software Inc.

[Don92] Donald Becker *A general non-shared-memory NS8390 ethernet driver for linux: Source Code.* Goddard Space Flight Center, NASA

[Don93] Donald Becker *A WD80x3 ethernet driver for linux: Source Code.* Goddard Space Flight Center, NASA